© COPYRIGHT 2005 by The MathWorks, Inc.

Tips for Fixed-Point Modeling and Code Generation
Release 14 with Service Pack 1

By Vinod Reddy and Siva Nadarajah

Introduction

This article provides tips on performing fixed-point modeling and generating code from such modeling. Fixed-point technology brings with it unique difficulties and challenges. These tips are intended to help you

- Reduce development time and therefore reduce cost
- Reduce ROM, RAM, and execution-time requirements of the generated code
- Improve accuracy of results of fixed-point operations

Fixed-Point Summary

Fixed-point representation allows you to express a real number as an integer by specifying its word length (in bits) and the location of its binary point, as desired. See the example below, which represents the base-10 number +6.5 in fixed-point notation as an eight-bit data type.

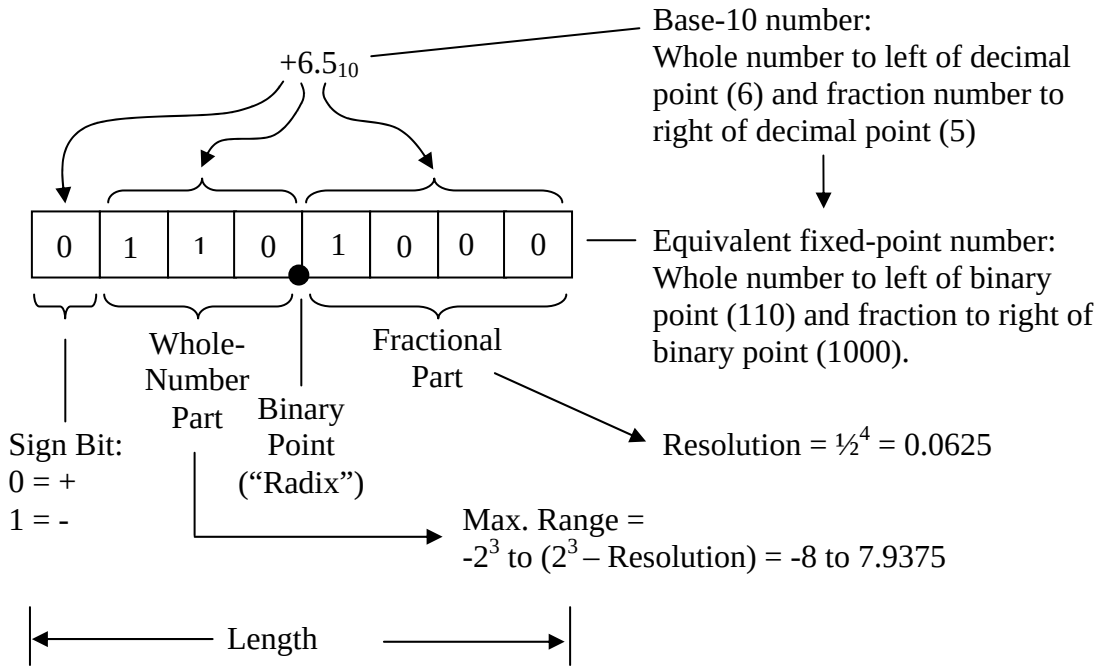

+6.5₁₀
Base-10 number:
Whole number to left of decimal point (6) and fraction number to right of decimal point (5)
0 1 1 0 1 0 0 0
Equivalent fixed-point number:
Whole number to left of binary point (110) and fraction to right of binary point (1000).
Whole-Number Part
Binary Point ("Radix")
Fractional Part
Sign Bit:
0 = +
1 = -
Resolution = ½⁴ = 0.0625
Max. Range =
-2³ to (2³ – Resolution) = -8 to 7.9375
Length

Figure 1: Conversion of base-10 to fixed-point.

Converting the whole-number part of the decimal number to fixed-point is more straightforward than converting the fractional part. Here is how to convert an entire decimal number to its equivalent fixed-point number:

- The whole-number part of the fixed-point number is the binary equivalent of the decimal whole number. In our example, the 6 converts to 110.
- The fractional part of the fixed-point number is the binary equivalent of the decimal fraction divided by the resolution. The resolution is $1/2^E$, where E is the number of bits to the right of the

© COPYRIGHT 2005 by The MathWorks, Inc.

binary point. In our example, the resolution is $\frac{1}{2}^4 = 0.0625$. Therefore, the fractional part of the fixed-point number is the binary equivalent of 0.5/0.0625, or 1000.

The fraction determines the *resolution*, which is the smallest nonzero value that the fixed-point number can represent. (In MathWorks documentation, resolution is called *precision*.) The whole number determines the maximum *range* of the fixed-point number, namely -2^x to (2^x – resolution), where x is the number of bits to the left of the binary point (minus any sign bit). Note that changing the location of the binary point in a fixed-point number causes a tradeoff between range and resolution.

Additional terms in fixed-point contexts are *scaling*, *bias*, and *slope*. The slope and bias together make up the scaling of a fixed-point number. The location of the binary point changes scaling. Think of the familiar y = mx+b, where m is slope and b is bias. The terminology used for this in the MathWorks Simulink® Fixed Point documentation is V = S*Q + B, where Q is "quantized fixed-point value" or "stored integer," V is "real-world" (that is, base-10) value, S is the user-specified slope, and B is the user-specified bias.

Fixed Point vs. Floating Point

Using fixed-point or floating-point data types in modeling has different advantages and disadvantages, as seen in the comparison below:

Consideration	Fixed Point	Floating Point
RAM and ROM consumption	Small	Large
Execution time	Faster	Slower
Word size and scaling	Flexible	Inflexible
Development time	Long	Short
Implementation complexity	More	Less
Static Determinism	High. Fixed-point operations produce results using static range and resolution	Low. Floating-point operations produce results using dynamic range and resolution.
Error Prone	High. More prone to quantization and overflow errors due to smaller range	Low. Can easily produce good resolution and range
Hardware power consumption	Low.	High

Drawbacks of using fixed point

As noted in the above table, the major drawbacks of using fixed-point data types are

- Quantization errors due to limited dynamic range
- Longer development time
- Implementation complexity that can lead to errors

What can we do about the drawbacks?

- Quantization errors can be reduced by selecting appropriate scaling or resolution (position of binary or radix point) and word length or range (limited by hardware).
- Longer development time can be mitigated by using Model-Based Design, which helps to reduce development time and resources
- Implementation errors can be reduced by using simulation and rapid prototyping, in-loop testing, automated verification and automated scaling tools help reduce and detect errors

© COPYRIGHT 2005 by The MathWorks, Inc.

Choosing slope and bias

You must choose fixed-point slope S and bias B so that Q, quantized fixed-point value or stored value, relates to V, the real-world value, with reasonable range and resolution for the application. This is illustrated in the following graphs.

Note: The fixed-point number represented in the following graphs is sfix5_En2, with a range of –4 to 3.75, and a resolution of 0.25.

On this and the following graphs, the topwaveform is V. The other waveform is Q. The y-axis is the value of Q or V. The x-axis is time.

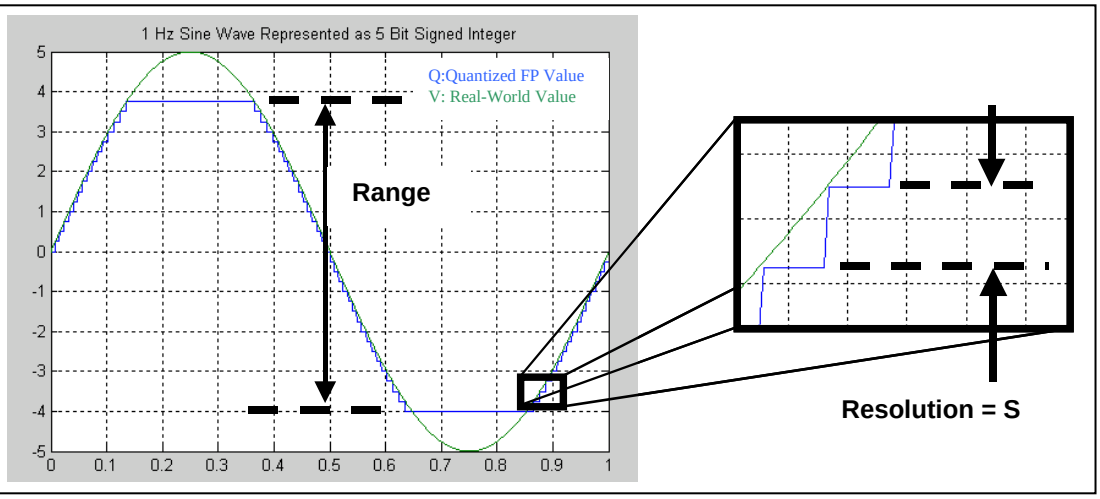


Figure 2: Example of range and resolution.

Note: The example above shows a five-bit integer for illustration purposes only. A more typical example is a 16- or 32-bit integer.

- You must adjust bias B to account for fixed biases between Q and V, which affects the range.

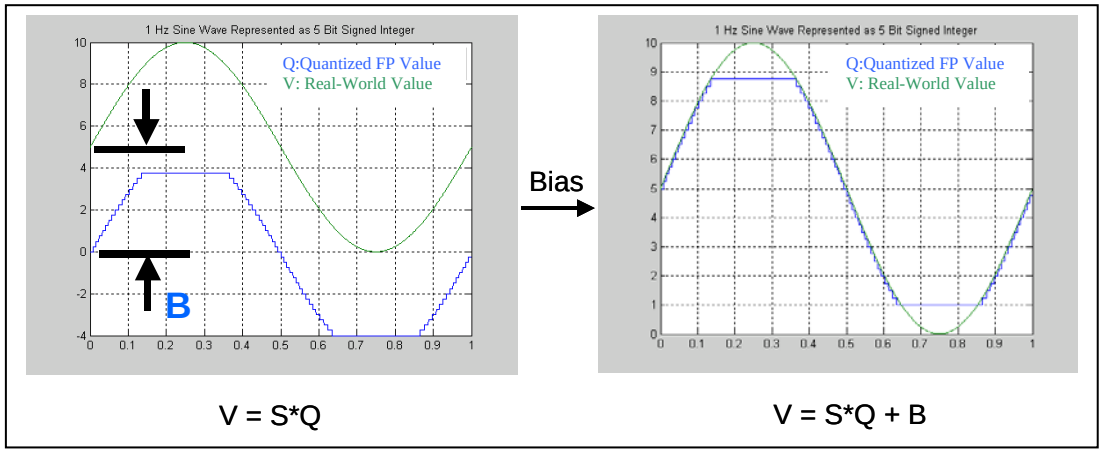

1 Hz Sine Wave Represented as 5 Bit Signed Integer
Q:Quantized FP Value
V: Real-World Value
B
Bias
1 Hz Sine Wave Represented as 5 Bit Signed Integer
Q:Quantized FP Value
V: Real-World Value
V = S*Q
V = S*Q + B

Figure 3: Adjusting bias.

© COPYRIGHT 2005 by The MathWorks, Inc.

Choosing a rounding method

You can choose a rounding method. When the target microcontroller converts a fixed-point number from a higher resolution to a lower resolution, there are additional bits to deal with. If the microcontroller discards the extra bits, the conversion can yield imprecise results. This is what occurs when you select the "floor" rounding method. However, floor is the simplest, most common, and most efficient of the rounding methods. Alternatively, you can select other rounding methods to preserve the extra bits for more accuracy.

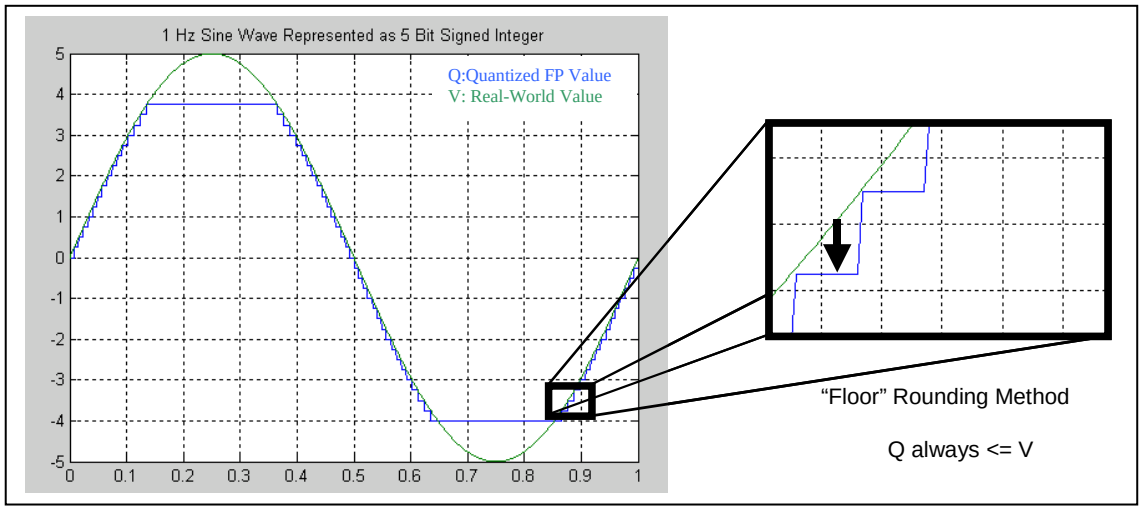

1 Hz Sine Wave Represented as 5 Bit Signed Integer
Q:Quantized FP Value
V: Real-World Value
"Floor" Rounding Method
Q always <= V

Figure 4: Example of floor rounding method.

Below are results from using different rounding methods for a resolution change from 0.125 to 0.5:

Value of Q with Resolution of 0.125	Value of Q with Resolution of 0.5 after Indicated Rounding Method*			
	Floor (- inf)	Ceiling (+ inf)	Zero (0)	Nearest
2.125	2.0	2.5	2.0	2.0
-2.125	-2.5`	-2.0	-2.0	-2.0
1.875	1.5	2.0	1.5	2.0
-1.875	-2.0	-1.5	-1.5	-2.0

* Terms in parentheses (-infinity, +infinity, and 0) indicate towards which the rounding is taking place.

© COPYRIGHT 2005 by The MathWorks, Inc.

Choosing an overflow handling method

You can choose an overflow handling method. “Wrap” and “saturate” are available. Overflow occurs when the number to be stored as a result of a mathematical operation exceeds the number of bits the resulting data type can accommodate.

Note: Use saturate only if the algorithm requires it, because it increases ROM and execution time.

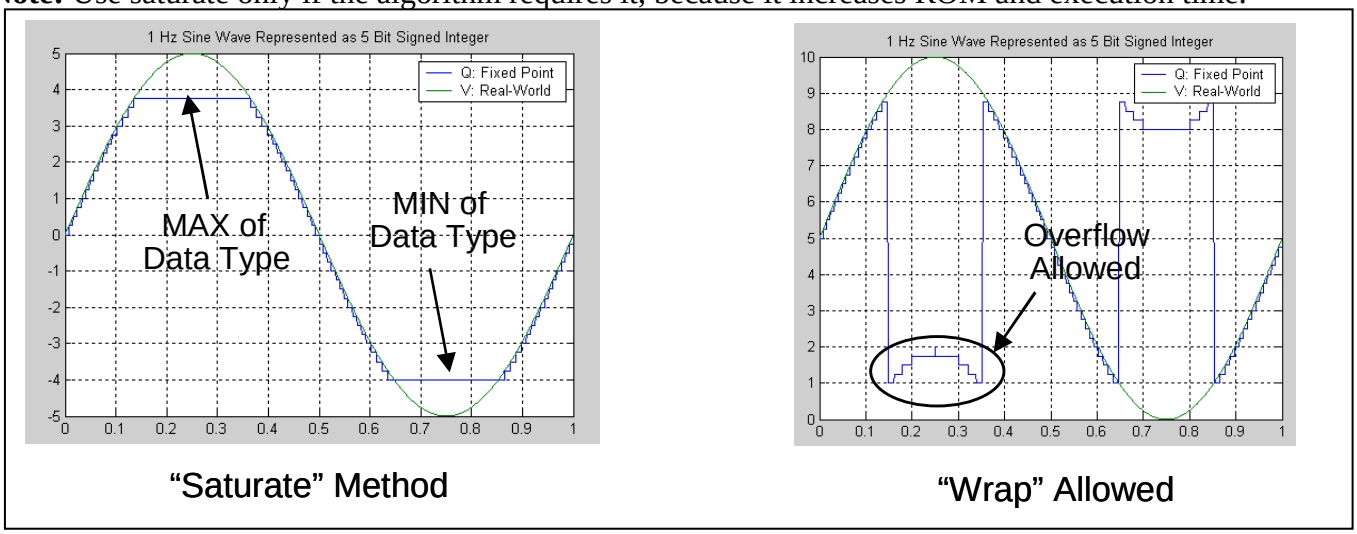

1 Hz Sine Wave Represented as 5 Bit Signed Integer
Q: Fixed Point
V: Real-World
MAX of Data Type
MIN of Data Type
“Saturate” Method
1 Hz Sine Wave Represented as 5 Bit Signed Integer
Q: Fixed Point
V: Real-World
Overflow Allowed
“Wrap” Allowed

Figure 5: Comparison of handling methods.

For each block output in the model, you can specify the word length and binary-point location. Depending on the kind of mathematical calculation performed on two or more fixed-point numbers, the result may require a longer word length. First, you need to simulate the model looking for outputs that best approximate values obtained from a floating-point simulation of the same model. An inappropriate tradeoff between range and resolution can cause an inaccurate result or unnecessary code size. Worse, an incorrect tradeoff can cause an erroneous result due to underflow or overflow. For additional information on this, refer to “Arithmetic Operations” in the Simulink Fixed Point documentation.

© COPYRIGHT 2005 by The MathWorks, Inc.

Simulink Fixed Point

Simulink Fixed Point replaces Fixed-Point Blockset and requires the Fixed-Point Toolbox. Simulink Fixed Point provides functionality, not blocks. All the blocks that were located in the Fixed-Point blockset library are now located in the Simulink library. Blocks with common functionality are now unified. Blocks with distinct functionality are absorbed.

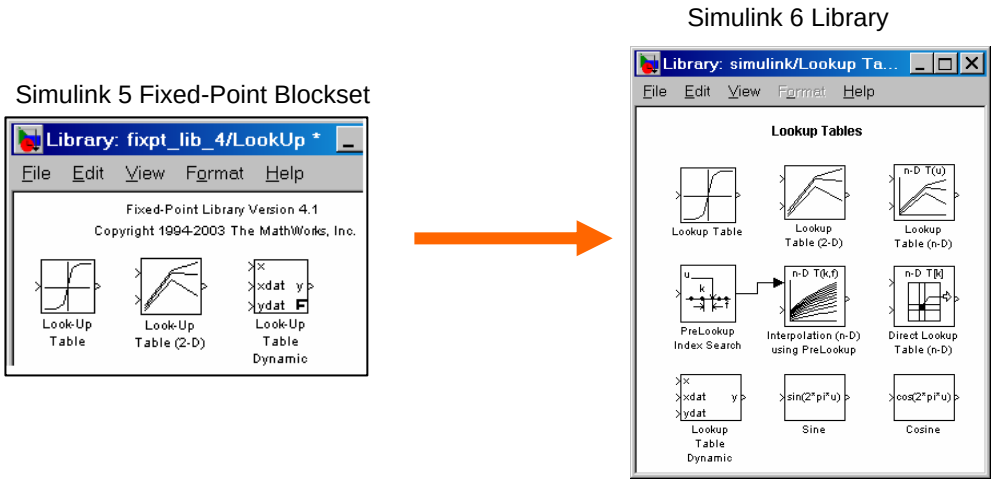
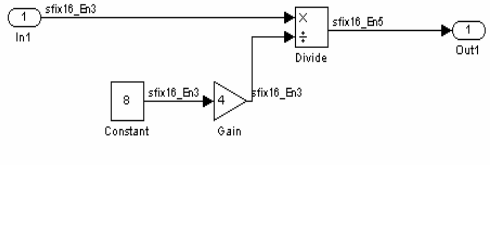


Figure 6: Fixed-Point Blockset unification in Simulink 6

In addition to the unification, several fixed-point operations have been optimized. The following code, using the rtwdemo_fixptdiv model, illustrates one such optimization. Note that multiple operations including the final scale are combined as a single shift right,




```
rtY.Out1 = (int16_T)(rtU.In1 >> 3);
```

Simulink 5 code

```
void DIV_S16_S16_S16_SL5_FLOOR(int16_T *C, int16_T A, int16_T B)
{
  if ( (B) == 0) {
    /* saturate to maximum or minimum */
   (*C) = (int16_T)( (A) >= 0 ? ((0x7FFF)) : ((0x8000U)) );
  } else {
    int quotientNeedsNegation = (A) < 0 != (B) < 0;
    unsigned int absA = (unsigned int)( (A) < 0 ? -(A) : (A) );
    unsigned int absB = (unsigned int)( (B) < 0 ? -(B) : (B) );
    (*C) = (int16_T)( ( absA << 5 ) / absB );
    if (quotientNeedsNegation) {
      if ( ( ( absA << 5 ) % absB ) > 0 ) {
       (*C)++;
      }
    }
    if (quotientNeedsNegation) {
     (*C) = -1 * (*C);
    }
  }
}
```

Simulink 6 code

Figure 7: Code generation comparison

When loaded in R14, the blocks in legacy models automatically map to the proper block in the Simulink library. This is seamless and fully automatic. This is true for both legacy fixed-point blocks and legacy Simulink blocks. The licensing rules introduced in R13 still apply. A Simulink Fixed Point license is required only when active use (fixed-point behavior in simulation or code generation) of fixed-point data types or fixed-point instrumentation occurs. To determine if a fixed-point license is installed, type `ver` at the MATLAB® command line.

© COPYRIGHT 2005 by The MathWorks, Inc.

Working with fixed-point models without a Simulink Fixed Point license

- You can edit a model containing fixed-point blocks, but you can not
  - Update or run a Simulink diagram using fixed-point behavior
  - Log minimum and maximum values produced by simulation
  - Automatically scale the outputs of a model using autoscaling tool

- You can simulate a model containing fixed-point blocks without a Simulink Fixed Point license:
  - Launch the Fixed-Point Settings interface by selecting **Tools->Fixed-Point Settings**:
  - Set the **Logging mode** parameter to `Force off model wide`
  - Set **Data type override** parameter to `True doubles or True singles`

© COPYRIGHT 2005 by The MathWorks, Inc.

Specifying data types in Simulink Fixed Point

When choosing a data type, you must consider the numerical range, resolution (precision), quantization error and a method for dealing with exceptional arithmetic conditions. The choices depend on the requirements of your specific application (accuracy required, response time), your embedded target (word size, processor speed, etc), and other factors.

With Simulink Fixed Point, you can explore the relationships among data types, range, precision, and quantization errors when modeling dynamic digital systems. With Real-Time Workshop®, you can generate and deploy production code based on that model.

The following are fixed-point data types in Simulink and Stateflow®:

- Generalized fixed-point types: ufix, sfix
  - There is no default binary point.
  - Binary point can be adjusted.
- Integer types: uint, sint
  - Binary point is to the right of the least significant bit.
  - Resolution is one.
  - Are compatible with built-in integer types (uint16, sint16…).
- Fractional types: ufrac, sfrac
  - Binary point is to the left of the most significant bit.
  - Resolution is 2^{-ws} for ufrac and $2^{-(ws-1)}$ for sfrac.

As shown in Figure 8, set the data type mode to `Specify via dialog` to enable entry of a fixed-point type.

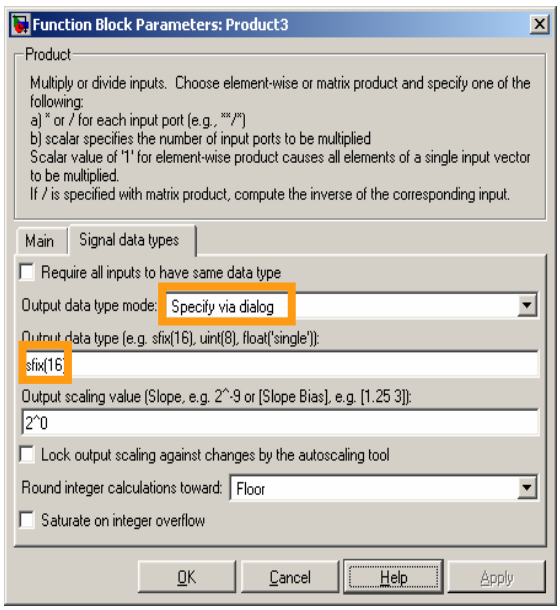

Function Block Parameters: Product3
Product
Multiply or divide inputs. Choose element-wise or matrix product and specify one of the following:
a) * or / for each input port (e.g., **/*)
b) scalar specifies the number of input ports to be multiplied
Scalar value of '1' for element-wise product causes all elements of a single input vector to be multiplied.
If / is specified with matrix product, compute the inverse of the corresponding input.
Main
Signal data types
Require all inputs to have same data type
Output data type mode:
Specify via dialog
Output data type (e.g. sfix(16), uint(8), float('single')):
sfix(16
Output scaling value (Slope, e.g. 2^-9 or [Slope Bias], e.g. [1.25 3]):
2^0
Lock output scaling against changes by the autoscaling tool
Round integer calculations toward:
Floor
Saturate on integer overflow
OK
Cancel
Help
Apply

Figure 8: Specify fixed-point data types

© COPYRIGHT 2005 by The MathWorks, Inc.

Specifying scaling methods in Simulink Fixed Point

Simulink Fixed Point supports two scaling methods: Binary point-only and [Slope Bias].

In Simulink: Specify the output scaling by entering 2^E (Figure 9) using binary point-only scaling method or [S B] (Figure 10) using the [Slope Bias] scaling method. The fixed-point settings of the block can be used to autoscale the output signal if it is not locked. Make sure the **Lock output scaling against changes by the autoscaling tool** option is not selected as shown in Figure 9.

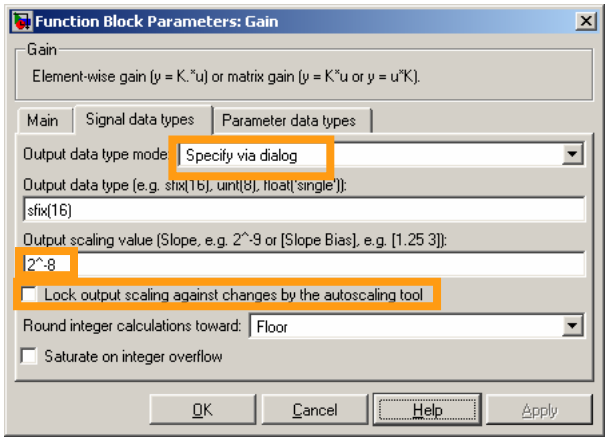

Function Block Parameters: Gain
Gain
Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).
Main
Signal data types
Parameter data types
Output data type mode: Specify via dialog
Output data type (e.g. sfix(16), uint(8), float('single')):
sfix(16)
Output scaling value (Slope, e.g. 2^-9 or [Slope Bias], e.g. [1.25 3]):
2^-8
Lock output scaling against changes by the autoscaling tool
Round integer calculations toward: Floor
Saturate on integer overflow
OK
Cancel
Help
Apply

Figure 9: Binary point-only scaling method in Simulink

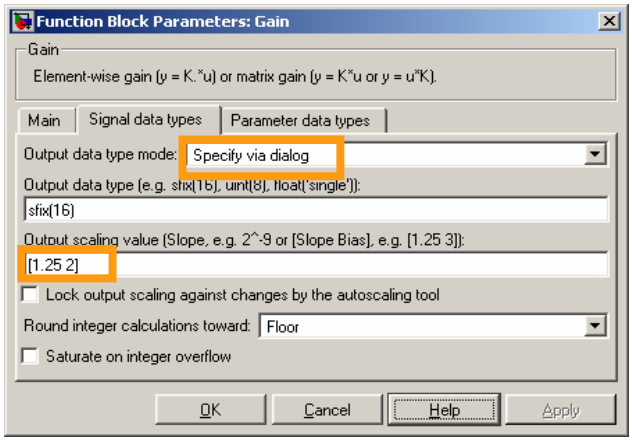

Function Block Parameters: Gain
Gain
Element-wise gain (y = K.*u) or matrix gain (y = K*u or y = u*K).
Main
Signal data types
Parameter data types
Output data type mode: Specify via dialog
Output data type (e.g. sfix(16), uint(8), float('single')):
sfix(16)
Output scaling value (Slope, e.g. 2^-9 or [Slope Bias], e.g. [1.25 3]):
[1.25 2]
Lock output scaling against changes by the autoscaling tool
Round integer calculations toward: Floor
Saturate on integer overflow
OK
Cancel
Help
Apply

Figure 10: Slope bias scaling method in Simulink

In Stateflow: Select `fixpt` from the **Type** pull-down list to enable entry of fixed-point type characteristics. Select the base type from the **Store integer** list to specify word size and signed/unsigned and then choose `Fraction length` to specify scaling using binary point-only method or `Scaling` to specify scaling using slope bias method.

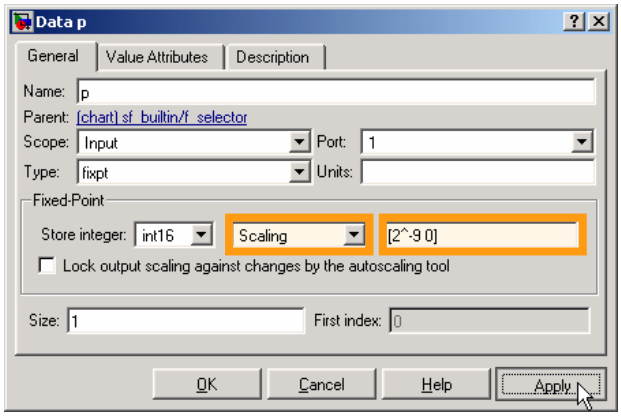


Figure 11: Slope bias scaling method in Stateflow

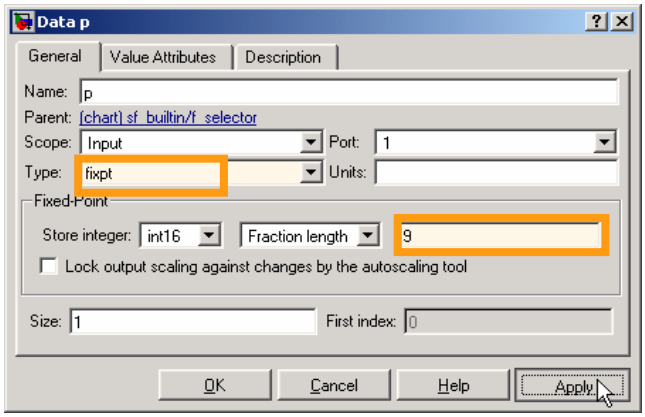


Figure 12: Binary point-only scaling method in Stateflow

© COPYRIGHT 2005 by The MathWorks, Inc.

Specifying rounding methods in Simulink Fixed Point

Simulink Fixed Point supports four rounding methods: Floor, Ceiling, Zero, and Nearest. Floor is the simplest, most common and efficient. Figure 13 shows the choice of rounding methods available for rounding integer calculations.

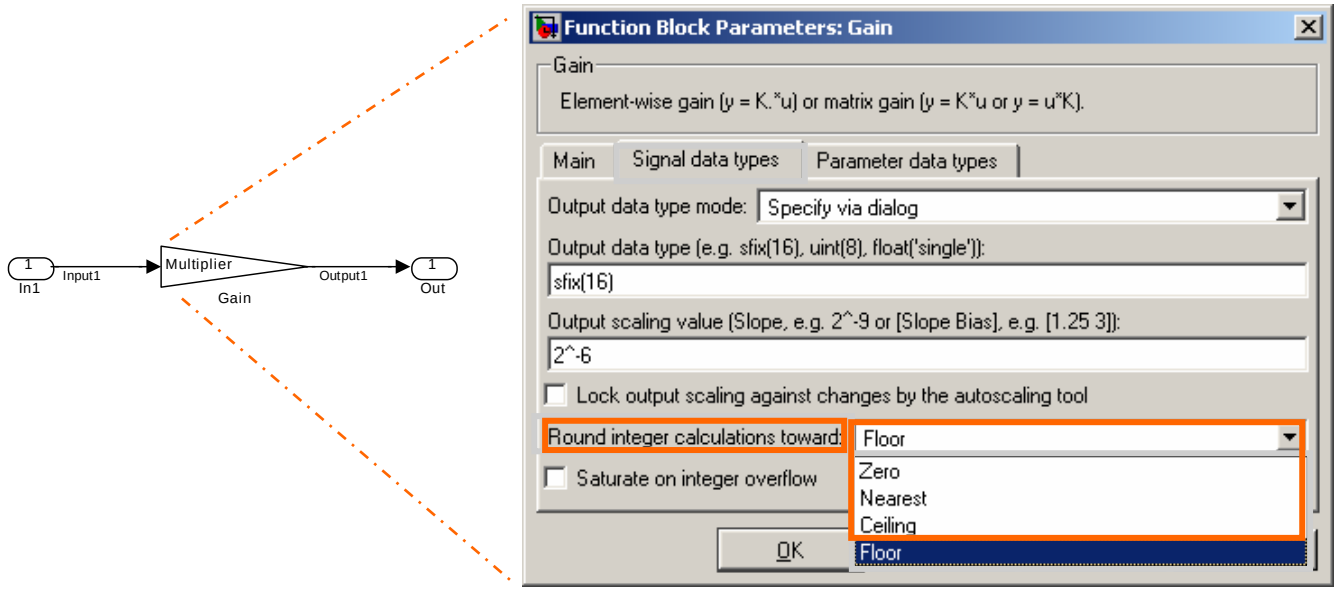
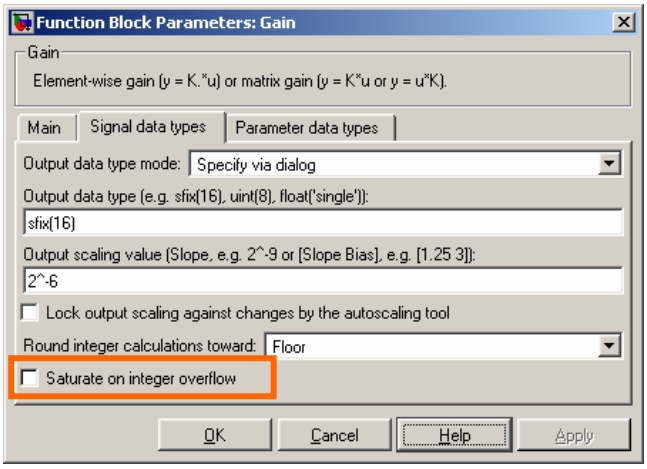


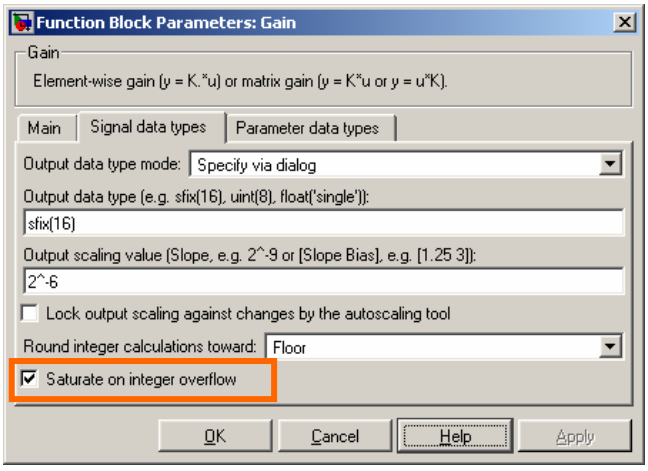
Figure 13: Rounding Methods

Specifying overflow handling methods in Simulink Fixed Point

Simulink Fixed Point support two methods to handle overflows: Wrap and saturate. The saturate method should be used only if it is essential for safety as it increases ROM and execution time.



Wrap on overflow



Saturate on overflow
(requires additional instructions/code)

Figure 14: Overflow Handling Methods

© COPYRIGHT 2005 by The MathWorks, Inc.

General Design Considerations

In designing a high-volume, low-cost embedded system, the general design considerations are cost, time to market, efficiency of the system to meet the timing constraints (execution time), accuracy, and the ability to reuse existing legacy software. .. Let us review these design considerations one at a time:

1. Cost
   - Minimize code and constants (ROM) size
   - Minimize execution time
   - Minimize data (RAM) size
   - Recurring vs. nonrecurring expense tradeoff
     - Hardware cost (Floating-point hardware is expensive.)
     - Engineering cost (Floating-point software is easier to design.)

2. Time to market
   - Minimize development time

3. Accuracy
   - Limited range and resolution of the fixed-point representation of real-world numbers will introduce output errors. Some applications are more sensitive to these errors than others.

4. Interface constraints
   - Interface with legacy code: hardware drivers, software libraries, etc.

Tips to Implement the General Design Considerations

The optimum solution considers both the design and implementation needs of the development process, and does so throughout the process. A design perceived as "stellar" during the modeling stage is not satisfactory if, for example, the implemented code consumes too much ROM or RAM, executes too slowly, or yields inaccurate results. The collection of tips below condenses years of experience in performing fixed-point modeling and implementing its generated code in embedded microcontrollers. These tips take into account both modeling and implementation. Knowing these tips, and incorporating them at strategic locations identified in the preceding table, will optimize your overall design.

The following sections discuss all of the design considerations in detail:

- General solution
- Designing for reduction of ROM
- Designing for reduction in execution time
- Designing for reduction of RAM
- Designing for accuracy
- Designing for interfaces
- Other modeling tips

The tips apply to MathWorks Release 14 with Service Pack 1. In the code examples used, the comments feature of Real-Time Workshop Embedded Coder was turned off for space considerations. All `.mdl` and `.m` files referenced in this document are available.

© COPYRIGHT 2005 by The MathWorks, Inc.

General Solution

The MathWorks provides tools that aid in the development and testing of dynamic systems. If your embedded target hardware is only fixed-point capable, you may design a dynamic system using floating-point numbers at first and then convert to a model with fixed-point numbers. Alternatively, you can start with the fixed-point design and skip the floating-point design entirely. Once the design requirements are met, you can generate code and deploy it on to your embedded target. A general development cycle is shown in Figure 15.

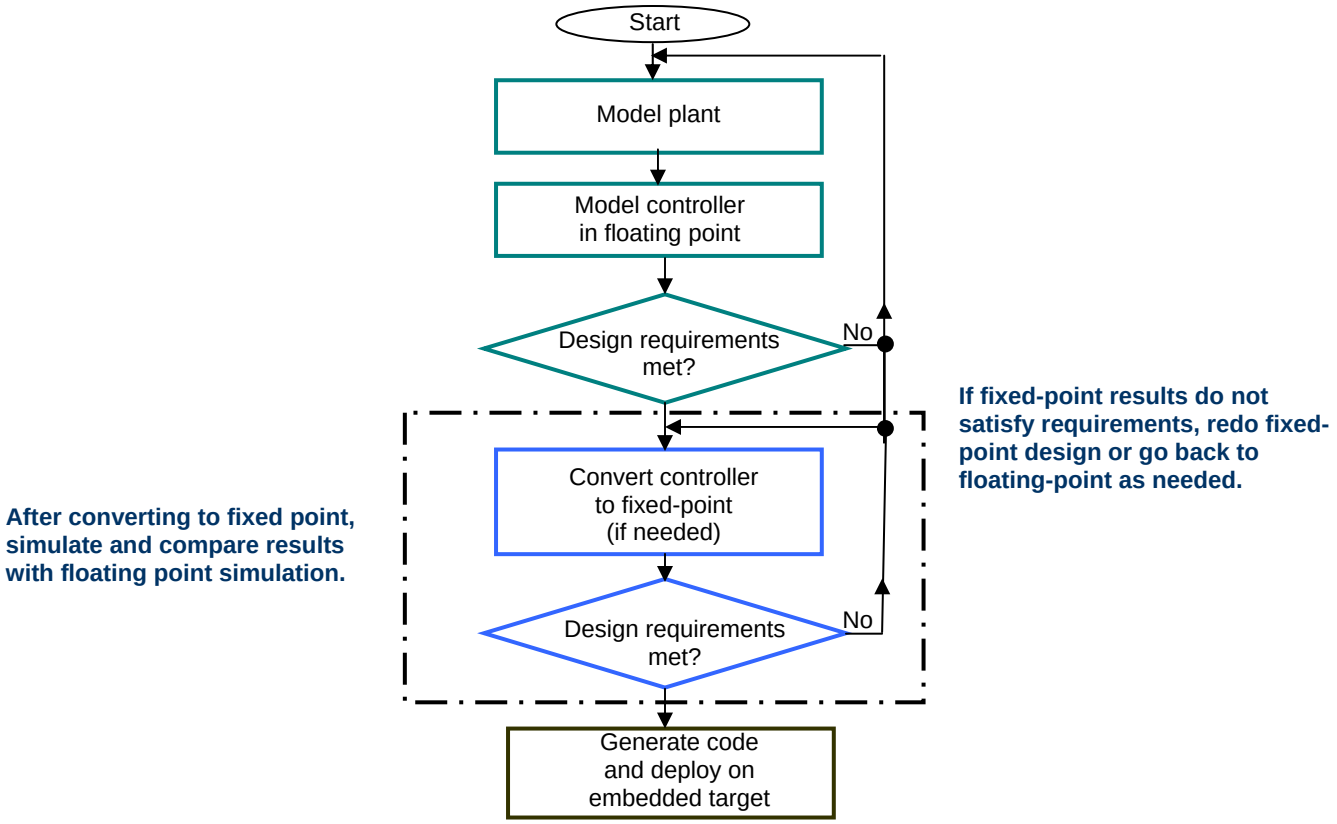

Start
Model plant
Model controller in floating point
Design requirements met?
No
If fixed-point results do not satisfy requirements, redo fixed-point design or go back to floating-point as needed.
Convert controller to fixed-point (if needed)
After converting to fixed point, simulate and compare results with floating point simulation.
Design requirements met?
No
Generate code and deploy on embedded target

Figure 15: General development cycle

The following table describes the steps involved in a general solution to fixed-point modeling. This is a development cycle often employed in various industries that use Model-Based Design ,such as manufacturing. Implementing the recommended tips where indicated in this general solution will considerably improve your process.

© COPYRIGHT 2005 by The MathWorks, Inc.

Steps in the General Solution	**Recommended Tip** (See explanations in *Tips for Optimizing the General Solution*)
I. Develop a floating-point model of the control algorithm using blocks from the Simulink block libraries	
II. Validate that the algorithms are correct in the floating-point model by performing simulation or rapid prototyping.	
III. Convert the floating-point model to fixed-point.	
A. If the model contains continuous blocks, replace with the equivalent discrete blocks.	
B. Establish the fixed-point representation.	
1. In R14, many Simulink blocks support fixed-point intrinsically.	Tip 24
2. Specify fixed-point data types and scaling for Simulink blocks.	Tip 21, 25, See Note
3. If model contains Stateflow charts with fixed-point data, use Model Explorer to specify fixed-point data types and scaling.	Tip 21, 25, See Note
4. Bring data into the simulation or export it from the simulation via inports and outports.	Tip 23
C. Choose integer sizes using Hardware Implementation Pane in Configuration Parameters dialog box	Tip 26
D. If you are unsure what the ranges and resolutions of the model's outputs should be, use the Autoscale Blocks button on the Fixed-Point Settings dialog box. If you want to override all the Simulink and Stateflow data types as a double data type during simulation, use the Data Type Override field in the Fixed-Point Settings dialog box.	Tip 29
E. Run simulation and inspect results by comparing them to floating-point results.	
F. Fine-tune the scaling and data type as needed for each block, and select the saturation and rounding method. Repeat steps D and E until the fixed-point design is acceptable.	Tip 5
IV. Validate the fixed-point design by performing simulation or rapid prototyping to ensure that the model's controller performance is within acceptable limits when compared with the floating-point simulation in step II above. If results are not within acceptable limits, return to step III.	
V. Generate the code.	
A. Review integer sizes based on the target microcontroller.	Tip 26
B. Run Model Advisor to optimize the Configuration Parameters setting.	Tip 28
C. Decide if generated code functions, data types, and packaging are acceptable.	Tips 7-11, 20-23, 27
D. Minimize the number of fixed-point types and scaling combinations to reduce the number of functions generated.	Tips 2, 3, 4
E. If you do change type and scaling combination settings, then you have changed the fixed-point design. Return to step III D or IV, as necessary.	
F. Inspect the generated code to determine additional optimization opportunities. For example, optimize blocks with cumbersome and unnecessary math operations. For this step, use the HTML Code Generation Report with links between the code and the model. For information on the report, see Real-Time Workshop Embedded Coder documentation.	Tips 1-19
G. Compile and link code to check RAM and ROM size. See the Real-Time Workshop Embedded Coder documentation on how to compile the generated code with MATLAB dependencies.	
VI. Deploy and validate the code on the ECU (possibly preceded by running it on an evaluation board).	
A. Validate the software execution results against the fixed-point design in step IV.	
B. Check execution timing against timing requirements.	

Note: Also refer to Specifying Data Types in Simulink Fixed Point and Specifying Scaling Methods in Simulink Fixed Point discussed in the preceding sections.

© COPYRIGHT 2005 by The MathWorks, Inc.

Designing for Reduction of ROM and Execution Time

The following tips can be used to design your model to reduce the ROM consumption and the execution time required to execute the Real-Time Workshop Embedded Coder generated code from the model:

- Tip 1. Use base data types to conserve ROM (ROM, execution time)
- Tip 2. Minimize mismatched data types (ROM, execution time)
- Tip 3. Minimize mismatched scaling (ROM, execution time)
- Tip 4. Avoid fixed-point scaling with bias (ROM, execution time)
- Tip 5. Round to floor and wrap (ROM, execution time)
- Tip 6. Optimize the order of multiply and divide operations (ROM, execution time)
- Tip 7. Limit the use of unevenly spaced lookup tables (ROM, execution time)
- Tip 8.Prevent evenly spaced lookup tables from being treated as unevenly spaced (ROM, execution time)
- Tip 9. Use power of two spaced breakpoints in lookup tables (ROM, execution time)
- Tip 10. Use reusable subsystems (ROM)
- Tip 11. Use shared utility feature (ROM)
- Tip 12. Use Stateflow operator ”C” (ROM, execution time)

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 1. Use base data types to conserve ROM

Restricting fixed-point data type word lengths so that they are equal to or less than the integer size of the target microcontroller reduces ROM and execution time. Thus, restricting data type word lengths results in fewer mathematical instructions in the target microcontroller. Otherwise, more mathematical instructions are required.

```
>> BaseDataTypes32
```

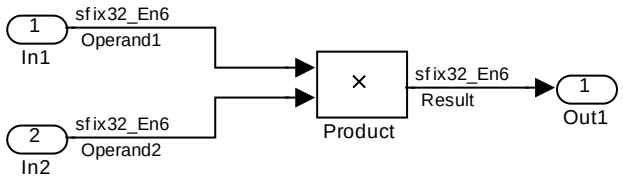


Figure 16: Model BaseDataTypes32.mdl

For example, as shown in Figure 16, performing a multiplication or division operation with 32-bit operands on a 16-bit microcontroller adds overhead in the code. Notice the code labeled "Added Code" shown in Figure 17.

```
int32_T Result;
int32_T mul_s32_s32_s32_sr6(int32_T a, int32_T b)
{
  uint32_T u32_chi;
  uint32_T u32_clo;
  mul_wide_s32(a, b, &u32_chi, &u32_clo);
  u32_clo = u32_chi << 26 | u32_clo >> 6;
  return (int32_T)u32_clo;
}

void BaseDataTypes32_step(void)
{
  Result = mul_s32_s32_s32_sr6(Operand1, Operand2);
}
```

Added Code

Figure 17: Code is added due to the data type word length being larger than the target integer size

```
>> BaseDataTypes16
```

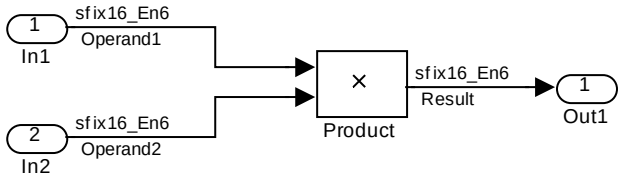


Figure 18: Model BaseDataTypes16.mdl

The model shown in Figure 18 uses 16-bit data types for the 16-bit target microcontroller. The generated code is as shown in Figure 19. Notice how clear and concise the code is when compared with that shown in Figure 17.

```
int16_T Result;

void BaseDataTypes16_step(void)
{
  Result = (int16_T)((int32_T)Operand1 * (int32_T)Operand2 >> 6);
}
```

Figure 19: More concise code due to the data type size being less than or equal to the target integer size

To reduce ROM and execution time:

1. Determine the integer size of the target microcontroller
2. Open the model
3. Restrict fixed-point data type sizes to be equal to or less than the integer size of the target microcontroller
4. Generate code

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 2. Minimize mismatched data types

When math blocks use common (matching) fixed-point data types, the generated math functions are reused.

In the model shown in Figure 20, there is a variety of fixed-point data types (`sfix8_En2, sfix16_En2, sfix24_En2`). As a result, the generated code, shown in Figure 21, contains a variety of fixed-point functions (div_s16s32, div_s16, DIV_MACRO_s16s24, LSL_S32, FIX2FIX_S24_S8_SL2) and only one function, div_s16s32, is reused.

```
>> mismatched_types
```

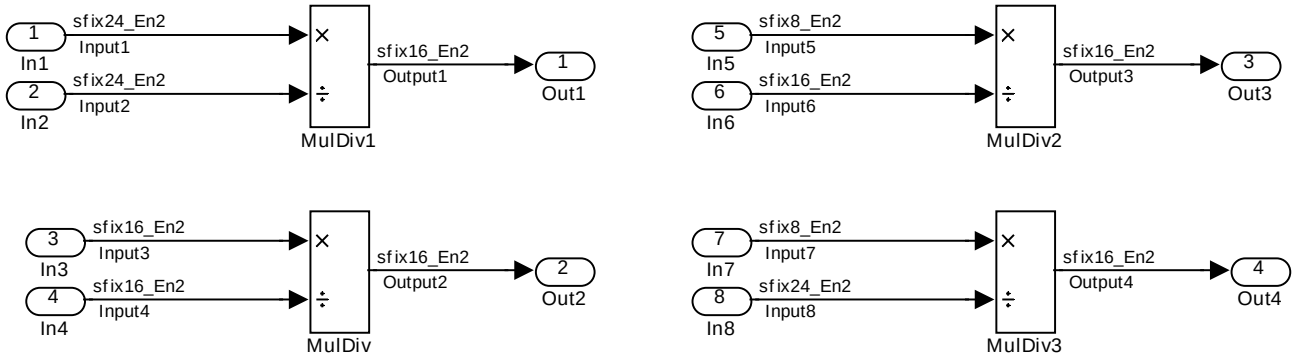


Figure 20: Model mismatched_types.mdl

```
void mismatched_types_step(void)
{

        Output1 = div_s16s32(LSL_S32(2,((int32_T)Input1)),((int32_T)Input2));

        Output2 = div_s16s32((int32_T)Input3 << 2, (int32_T)Input4);

        Output3 = div_s16(Input5 << 2, Input6);

        DIV_MACRO_s16s24(Output4,FIX2FIX_S24_S8_SL2(Input7),Input8);

}
```

Math functions: div_s16s32, div_s16, DIV_MACRO_s16s24, LSL_S32,FIX2FIX_S24_S8_SL2
div_s16s32 is reused

Figure 21: Extra math functions added due to mismatched data type sizes

To enable code reuse, restrict fixed-point data types in math functions to a small subset.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 3. Minimize mismatched scaling

When scaling is binary-point only , shifts (left/right) are sufficient to handle scaling adjustment. Situations where shifts are not sufficient to handle scaling are called mismatched scaling.

In the model shown in Figure 22 below, a multiplication by an integer correction term (multiplication by 21299L) in addition to shifting is needed to calculate the value of Upper. This is due to mismatched scaling (Input1 and Constant1 have mismatched slopes and the slope of Constant1 is an inexact power of two). As shown in Figure 23, a multiplication by an integer correction term is not required to calculate Lower as Input2 and Constant2 have matched scaling set to an exact of power of two.

```
>> mismatched_scaling
```

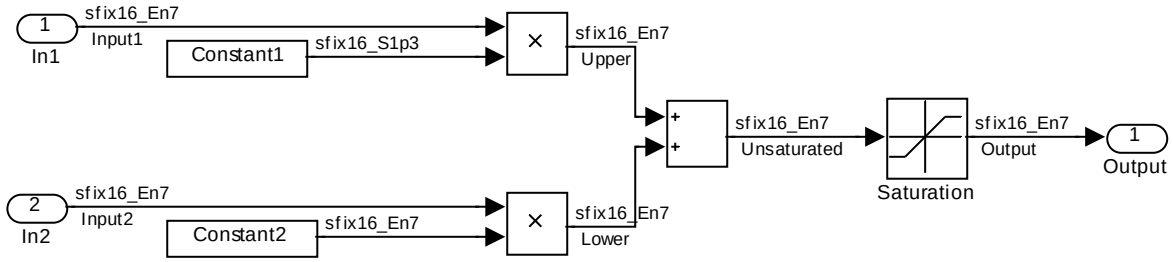


Figure 22: Model mismatched_scaling.mdl

```
void mismatched_scaling_step(void)
{
 int16_T rtb_Unsaturated;
 Upper = (int16_T)((int32_T)(int16_T)((int32_T)Input1 * (int32_T)Constant1) * 21299L >> 14);
 Lower = (int16_T)((int32_T)Input2 * (int32_T)Constant2 >> 7);
 rtb_Unsaturated = Upper + Lower;
 Output = rt_SATURATE(rtb_Unsaturated, -640, 2048);
}
```

Code for mismatched scaling

Code for matched scaling

Figure 23: Integer correction term code added for operation using mismatched scaling

To reduce both ROM and execution time, minimize the occurrences of mismatched scaling in your design.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 4. Avoid fixed-point scaling with bias

In most cases, mathematical operations involving fixed-point numbers with bias increase ROM consumption and execution time. In certain cases, if you select appropriate biases for the mathematical operations, you may be able to avoid these increases. For example, if you are performing an addition operation, the bias of the inputs should add up to the bias of the output.

Therefore, we recommend that you do not use g fixed-point numbers with bias unless it is essential. For example, if you are interfacing to hardware devices, using biases might be required. The hardware device may have a built-in bias. In such cases, you must have fixed-point numbers with bias in order to interface to the device correctly.

In the model shown below, the calculation of Upper contains multiplication of two signals (Input1, Constant1) that have their bias set to zero (no bias). The calculation of Lower contains multiplication of two inputs (Input2, Constant2), but one of the inputs is set to bias of 2. Notice the difference in the generated code for each of the cases.

```
>> mismatched_bias
```

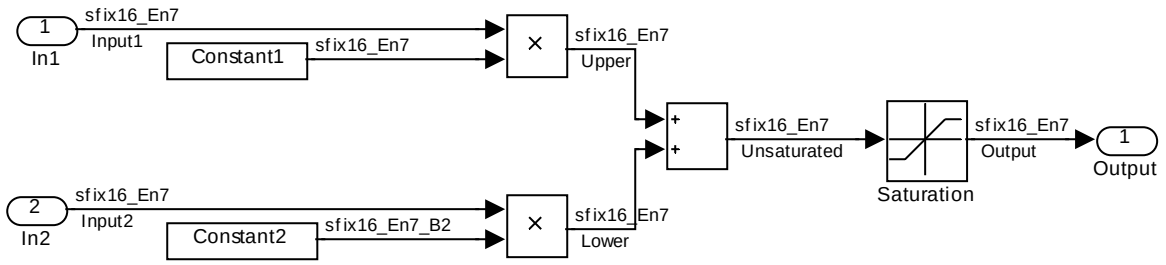


Figure 24: Model mismatched_bias.mdl

```
void mismatched_bias_step(void)
{
  int16_T rtb_Unsaturated;
  Upper = (int16_T)((int32_T)Input1 * (int32_T)Constant1 >> 7);
  {
    int32_T bTempMulNiceScaling;
    bTempMulNiceScaling = 256;
    bTempMulNiceScaling += ((int32_T)Constant2);
    MUL_S16_S32_S16_SR7(&(Lower),bTempMulNiceScaling,Input2);
  }
  rtb_Unsaturated = Upper + Lower;
  Output = rt_SATURATE(rtb_Unsaturated, -640, 2048);
}
```

Code for input without bias

Code for input with bias

Figure 25: Extra code is added for input with bias

Whenever possible avoid fixed-point scaling with bias. Additional information on this topic is in "Recommendations for Arithmetic and Scaling" section of Simulink Fixed Point documentation.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 5. Round to floor and wrap

Expression folding combines outputs of multiple sequential Simulink blocks into a single expression during code generation in order to reduce ROM and execution time. To obtain the full benefit of expression folding for each block in the model, clear the **Saturate on integer overflow** check box and select `Floor` in the **Round integer calculations toward** field on the Block Parameters dialog, as shown for the model below. Figure 28 shows the generated code.

Caution: Turn off saturation only if you are not concerned about overflow, and set the rounding option to Floor only if you do not need maximum numerical precision.

```
>> OverflowRounding
```

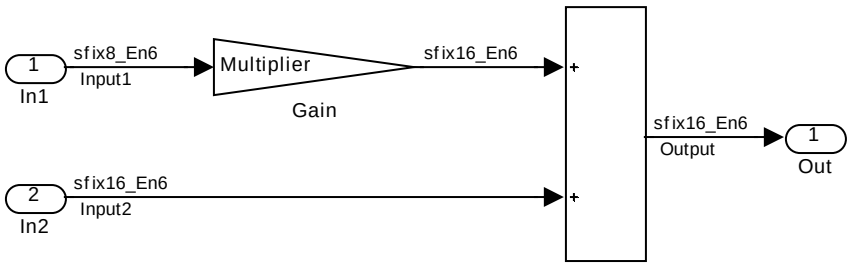


Figure 26 : Model OverflowRounding.mdl.

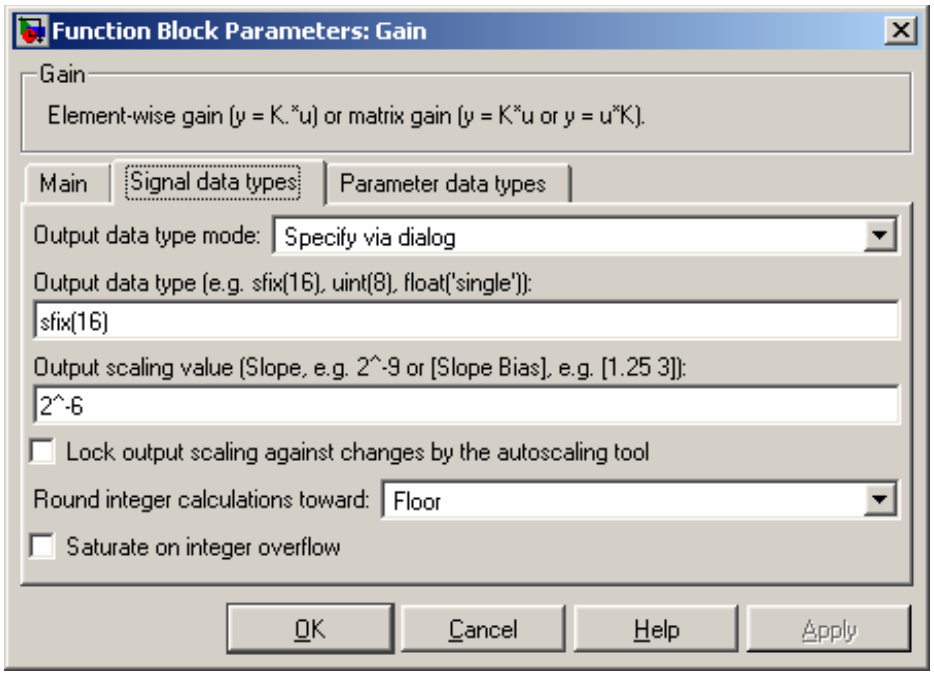


Figure 27: Overflow and rounding settings for Gain block.

© COPYRIGHT 2005 by The MathWorks, Inc.

```
int16_T Output;

const int8_T Multiplier = 77;

void OverflowRounding_step(void)
{
   Output = (Input1 * Multiplier >> 6) + Input2;
}
```

Figure 28: This code has neither overflow check nor rounding.

Note that code for the multiplication and addition blocks are combined into the single statement

Output = (Input1 * Multiplier >> 6) + Input2;

By comparison, Figure 29 shows the less efficient code that results from selecting **Round integer calculations toward** to be `Zero`.

```
int16_T Output;

const int8_T Multiplier = 77;

void OverflowRounding_step(void)
{
   {
     int16_T _fixptlowering0;

     _fixptlowering0 = Input1 * Multiplier;

     Output = ((_fixptlowering0 >> 6) +
      ((_fixptlowering0 < 0) && (_fixptlowering0 & 63))) + Input2;
   }
}
```

Figure 29: Code with rounding set to zero.

Figure 30 shows the less efficient code that results from selecting **Round integer calculations toward** to be `Zero`**, Saturate on integer overflow**, and changing the Gain block output to type `sfix(8)`.

© COPYRIGHT 2005 by The MathWorks, Inc.

```
int16_T Output;

const int8_T Multiplier = 77;

void OverflowRounding_step(void)
{
  {
     int16_T _fixptlowering0;
     int8_T _fixptlowering2;

     _fixptlowering0 = Input1 * Multiplier;
     _fixptlowering0 = (_fixptlowering0 >> 6) +
       ((_fixptlowering0 < 0) &&  (_fixptlowering0 & 63));
     if(_fixptlowering0 > 127) {
       _fixptlowering2 = MAX_int8_T;
     } else if(_fixptlowering0 <= -128) {
        _fixptlowering2 = MIN_int8_T;
     } else {
        _fixptlowering2 = (int8_T)_fixptlowering0;
     }
     Output = _fixptlowering2 + Input2;
  }
}
```

Figure 30: Code with overflow check, rounding set to zero and Gain block output data type set to `sfix(8)`.

For each block in the model:

1. Double-click the block. The Block Parameters dialog box appears. Select the **Signal data types** tab.
2. Select Floor in the **Round integer calculations toward** field. (`Floor` is the default.)
3. Clear **Saturate on integer overflow**. (`Clear` is the default.)
4. Generate code.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 6. Optimize the order of multiply and divide operations (ROM, execution time)

When a product block is configured with a divide operation for the first input and a multiply for the second, it results in a reciprocal operation followed by a multiply operation. Ensure that multiplication occurs first and then division. This results in a single division operation.

In the model shown in Figure 31 below, for the block labeled DivMul, where the first input is a divide and the second input is multiply, a reciprocal operation on Input1 is performed prior to multiplying it with Input2. For the block labeled MulDiv, where the first input is multiply and the second is divide, Input3 is simply divided by Input4, avoiding the reciprocal operation as shown in Figure 32.

```
>> order_mul_div
```

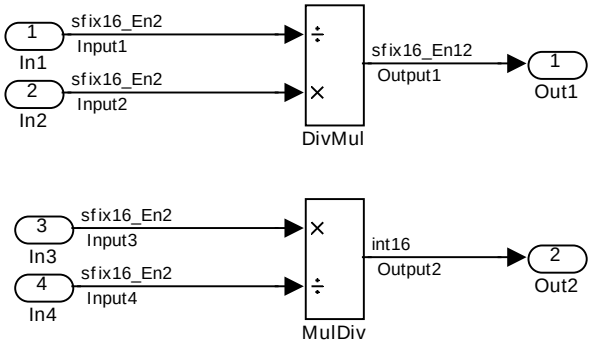


Figure 31: Model order_mul_div.mdl

```
#define DIV_MACRO_s16(quotient,numerator,denominator)   DIV_MACRO_s16 is not shown here for space considerations
#define ASRl(nBits,C) ( (C)>>(nBits) )
#define MUL_S16_S16_S16_SR2_ZERO(C,A,B) {C = FIX2FIX_S16_S32_SR2_ZERO((((long)(A)) * ((long)(B))))};

int16_T div_s16(int16_T numerator, int16_T denominator)
{
     int16_T quotient;
     uint16_T tempAbsQuotient;
     if(denominator == 0) {
       quotient = numerator >= 0 ? MAX_int16_T : MIN_int16_T;
     } else {
       tempAbsQuotient = (uint16_T)((uint16_T)(numerator >= 0 ? numerator :
           -numerator) / (uint16_T)(denominator >= 0 ? denominator : -denominator));
          return numerator < 0 != denominator < 0 ? (int16_T)-(int16_T)tempAbsQuotient
            : (int16_T)tempAbsQuotient;
     }
     return quotient;
}

int16_T FIX2FIX_S16_S32_SR2_ZERO(long B)
{
     return (((int16_T)( ASRl(2,B) + ( ( B < 0 ) && ( B & (0x0003) ) ) )));
}

void order_mul_div_step(void)
{
    DIV_MACRO_s16(Output1,(0x4000),Input1);

    MUL_S16_S16_S16_SR2_ZERO(Output1,Output1,Input2);

    Output2 = div_s16(Input3, Input4);

}
```

Additional code is required when the first input is divide and the second input is multiply

No additional code is required when the first input is mutliply and the second input is multiply

Figure 32: Code added when the first input of a product block is set to divide

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 7. Limit the use of unevenly spaced lookup tables (ROM, execution time)

The points along one or more independent input axes of a lookup table can be evenly spaced or unevenly spaced. There are advantages and disadvantages to each. An unevenly spaced input axis provides clustered points where necessary only at certain regions of the axis so as to produce greater lookup accuracy. As a result, the table can have fewer total points to achieve the same lookup accuracy compared with an evenly spaced table. However, unlike a table with one or more evenly spaced axes, a table with an unevenly spaced axis requires a search routine and memory for each input axis. This increases ROM and execution time.

Considering the needs of the algorithm and resources (execution time and ROM), decide whether you need an unevenly or evenly spaced table. Additional information on this is in the "Effects of Spacing on Speed, Error, and Memory Usage" section of the Simulink Fixed Point documentation.

Note: This tip applies only to tables with non-tunable input axes. For a table with tunable input axes, this tip provides no benefit. This is because tunable input axes always are treated as unevenly spaced.

Figures 34 and 35 compare code generated for an unevenly spaced table versus that generated for an evenly spaced table, for the model below.

```
>> Lookup
```

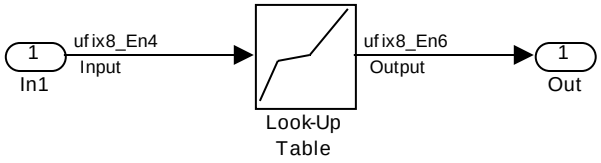


Figure 33: Model Lookup.mdl.

Figure 34 shows the generated code for uneven points along the x (input) axis:
x_breakpoints = [2.5 5 9.5 15]
y_outputs = [0 1.62 1.85 3.67]

Figure 35 shows the generated code for even points along the x (input) axis:
x_breakpoints = [0 5.0 10.0 15.0]
y_outputs = [0 1.62 1.85 3.67]

You can see that the code for the table with evenly spaced axes requires no search function and no memory for each input axis. Note that even better efficiency results if the input-axis spacing is 2^n, where n is any integer.

© COPYRIGHT 2005 by The MathWorks, Inc.

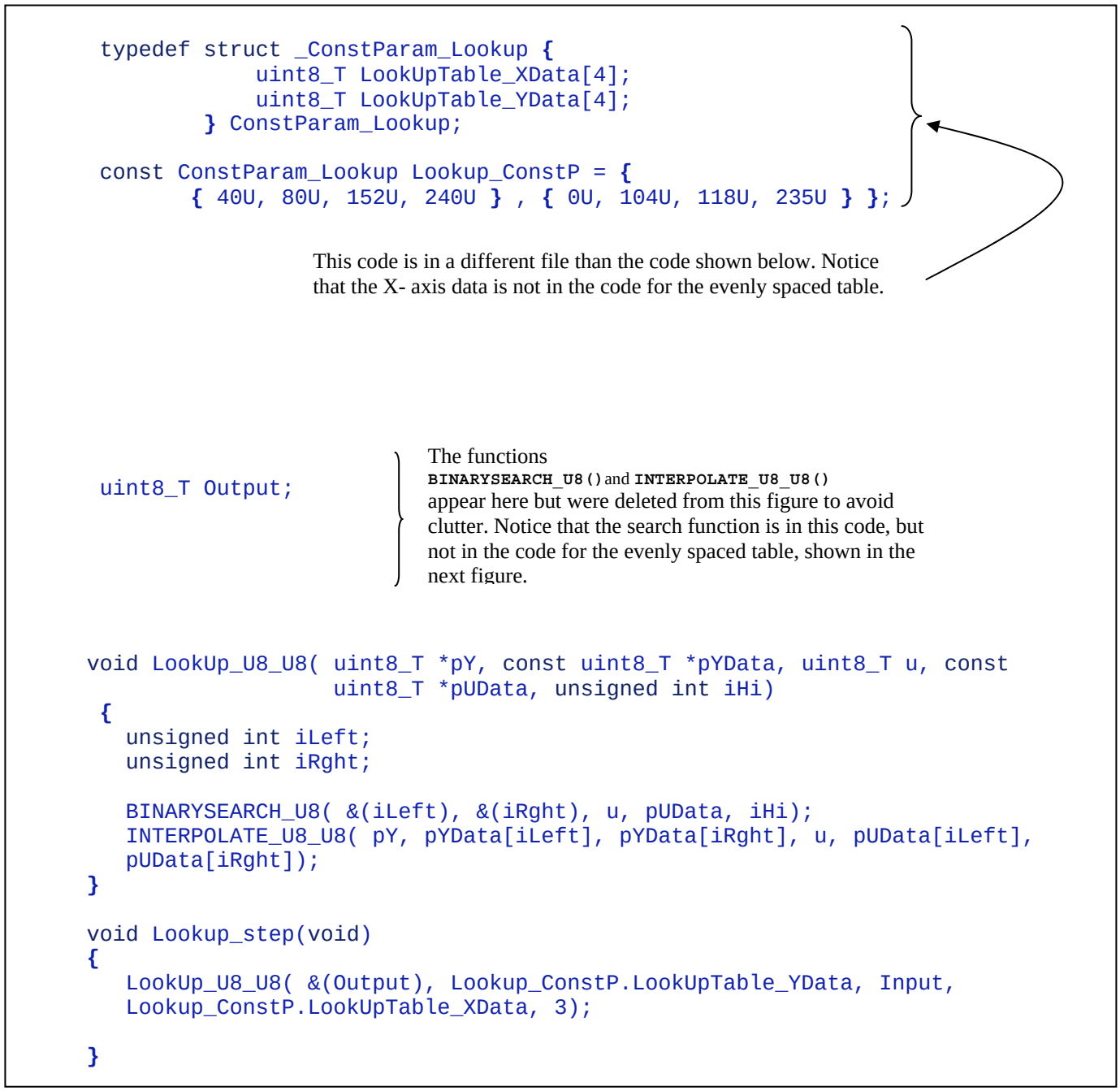

typedef struct _ConstParam_Lookup {
uint8_T LookUpTable_XData[4];
uint8_T LookUpTable_YData[4];
} ConstParam_Lookup;
const ConstParam_Lookup Lookup_ConstP = {
{ 40U, 80U, 152U, 240U } , { 0U, 104U, 118U, 235U } };
This code is in a different file than the code shown below. Notice that the X- axis data is not in the code for the evenly spaced table.
uint8_T Output;
The functions BINARYSEARCH_U8() and INTERPOLATE_U8_U8() appear here but were deleted from this figure to avoid clutter. Notice that the search function is in this code, but not in the code for the evenly spaced table, shown in the next figure.
void LookUp_U8_U8(uint8_T *pY, const uint8_T *pYData, uint8_T u, const
uint8_T *pUData, unsigned int iHi)
{
unsigned int iLeft;
unsigned int iRght;
BINARYSEARCH_U8(&(iLeft), &(iRght), u, pUData, iHi);
INTERPOLATE_U8_U8(pY, pYData[iLeft], pYData[iRght], u, pUData[iLeft],
pUData[iRght]);
}
void Lookup_step(void)
{
LookUp_U8_U8(&(Output), Lookup_ConstP.LookUpTable_YData, Input,
Lookup_ConstP.LookUpTable_XData, 3);
}

Figure 34: Code generated for unevenly spaced table

© COPYRIGHT 2005 by The MathWorks, Inc.

```
typedef struct _ConstParam_Lookup {
            uint8_T LookUpTable_YData[4];
        } ConstParam_Lookup;


const ConstParam_Lookup Lookup_ConstP = { { 0U, 104U, 118U, 235U } };
```

This code is in a different file than the code shown below. Notice that the x-axis data in the previous figure is not needed in this code.

The function **INTERPOLATE_EVEN_U8_U8()** appears here but was deleted from this figure to avoid clutter. Notice that the search function is not in this code, but in the code for the unevenly spaced

```
uint8_T Output;


void LookUpEven_U8_U8( uint8_T *pY, const uint8_T *pYData, uint8_T u,
        uint8_T  valueLo, unsigned int iHi, uint8_T uSpacing)
{

  if ( u <= valueLo )
  {
    (*pY) = (*pYData);
  }
  else
  {
    unsigned int uAdjusted = u - valueLo;
    unsigned int iLeft = uAdjusted / uSpacing;
    if ( iLeft >= iHi )
    {
      (*pY) = pYData[iHi];
    }
    else
    {
      INTERPOLATE_EVEN_U8_U8( pY, pYData[iLeft], pYData[((iLeft)+1)],
       ((uint8_T)((uAdjusted-(iLeft*uSpacing)))), uSpacing);
    }
  }
}


void Lookup_step(void)
{

  LookUpEven_U8_U8( &(Output), Lookup_ConstP.LookUpTable_YData, Input,
         0U, 3, 80U);

}
```

No need for Binary_Search_U8

Figure 35: Code generated for evenly spaced table

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 8. Prevent evenly spaced lookup tables from being treated as unevenly spaced

For this example, let X be the decimal number being converted to a binary fixed-point number, and Y be the resolution of the fixed-point number. If X/Y results in a whole number, there will be no quantization error. Otherwise, there will be a quantization error. Earlier we said that resolution is $1/2^E$. The greater the value of E, the smaller the quantization error.

Often, when Real-Time Workshop Embedded Coder translates an evenly spaced lookup table to a fixed-point lookup table, a quantization error results. That is, the points along an input axis of what began in the model as an evenly spaced lookup table are unevenly spaced in the generated code. This adds an x-axis data statement and a binary search routine to the generated code.

For additional information see “Effects of Spacing on Speed, Error, and Memory Usage” section of the Simulink Fixed Point documentation.

Note: This tip does not apply to a table with tunable input axis since the Real-Time Workshop creates a definition of the entire axis to allow for tunability of each point.

Observe the model below, for example.

```
>> LookupEvenBinSearch
```

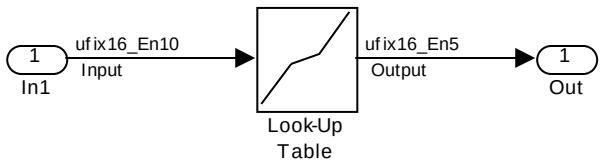


Figure 36: Model LookupEvenBinSearch.mdl

The lookup table has these settings:
x_breakpoints = [20.05 20.1 20.15 20.2]
y_outputs = [0 22.3 28.5 52.5]

Notice that the x-axis (input axis) has an evenly spaced point value of 0.05. Figure 37 shows the generated code. The fact that the code has an x-axis data statement and a search routine means that the x-axis points in the code are unevenly spaced.
Note: Instead of performing the solution below, we recommend first changing the resolution of the input signal to the lookup table. If this can be done without resulting in an unacceptable range of this input signal, the solution below may be unnecessary.

© COPYRIGHT 2005 by The MathWorks, Inc.

Here is the solution:

1. On the MATLAB command line, type the following function call:
   fixpt_evenspace_cleanup (x_axis, data_type, resolution), where x_axis is the name in the x-axis field of the Block Parameters dialog for the lookup table, data_type is the data type of the input signal to the lookup table, and resolution is the resolution of the input signal to the lookup table
   .
   For our example, you would type:

   ```
   fixpt_evenspace_cleanup (x_breakpoints, ufix(16), 2^-10)
   ```

   MATLAB displays new points for the x-axis. Our example results in
   `ans = 20.0498  20.0996  20.1494  20.1992`
2. On the MATLAB command line type `x_axis =[A B C D]`, where `x_axis` is the name of the array containing the breakpoints and the arguments A, B, C, and D are the new values that MATLAB displayed in the previous step. For our example, you would type
   `x_breakpoints = [20.0498  20.0996  20.1494  20.1992].`

   This redefines the x-axis points so that there will be negligible quantization error.

The code that results is shown in Figure 38 and is similar to that shown in the Figure 37. Notice there is no x-axis data statement and no search routine, indicating that the x-axis points in the code are evenly spaced.

```
typedef struct _ConstParam_LookupEvenBinSearch {
    uint16_T LookUpTable_XData[4];
    uint16_T LookUpTable_YData[4];
} ConstParam_LookupEvenBinSearch;

const ConstParam_LookupEvenBinSearch LookupEvenBinSearch_ConstP =
   {
     { 20531U, 20582U, 20634U, 20685U } ,  { 0U, 714U, 912U, 1680U }
   };


void LookUp_U16_U16( uint16_T *pY, const uint16_T *pYData, uint16_T u,
         const uint16_T *pUData, unsigned int iHi)
{
  unsigned int iLeft;
  unsigned int iRght;

  BINARYSEARCH_U16( &(iLeft), &(iRght), u, pUData, iHi);
  INTERPOLATE_U16_U16( pY, pYData[iLeft], pYData[iRght], u, pUData[iLeft],
   pUData[iRght]);
}


void LookupEvenBinSearch_step(void)
{

  LookUp_U16_U16( &(Output), LookupEvenBinSearch_ConstP.LookUpTable_YData,
        Input, LookupEvenBinSearch_ConstP.LookUpTable_XData, 3);
}
```

Figure 37: Evenly spaced table in model but unevenly spaced table in code.

© COPYRIGHT 2005 by The MathWorks, Inc.

```
typedef struct _ConstParam_LookupEvenBinSearch {
    uint16_T LookUpTable_YData[4];
} ConstParam_LookupEvenBinSearch;

const ConstParam_LookupEvenBinSearch LookupEvenBinSearch_ConstP =
   {
     { 0U, 714U, 912U, 1680U }
   };

void LookUpEven_U16_U16( uint16_T *pY, const uint16_T *pYData, uint16_T u,
 uint16_T valueLo, unsigned int iHi, uint16_T uSpacing)
{
  if ( u <= valueLo )
  {
    (*pY) = (*pYData);
  }
  else
  {
    unsigned int uAdjusted = u - valueLo;
    unsigned int iLeft = uAdjusted / uSpacing;
    if ( iLeft >= iHi )
    {
      (*pY) = pYData[iHi];
    }
    else
    {
      INTERPOLATE_EVEN_U16_U16( pY, pYData[iLeft], pYData[((iLeft)+1)],
       (uAdjusted-(iLeft*uSpacing)), uSpacing);
    }
  }
}


void LookupEvenBinSearch_step(void)
{
 LookUpEven_U16_U16( &(Output),
    LookupEvenBinSearch_ConstP.LookUpTable_YData, Input, 20531U, 3, 51U);
}
```

Figure 38: Evenly spaced table in model and evenly spaced table in code.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 9. Use power of two spaced breakpoints in lookup tables

Just like the evenly spaced lookup tables, power of two spacing also does not use breakpoints in the generated code. This is significantly better than unevenly and evenly spaced lookup tables. The key difference is that a subtraction and division are replaced by a bitwise AND combined with a shift right at the end of the multiplication. Interpolation is performed without loss of precision. Hence, power-of-two spaced breakpoints in lookup tables usually don't introduce rounding errors, require only Y data points, and are more accurate and is generally faster.

```
>> fxpdemo_approx_sin
```

For additional information, study the above demo (fxpdemo_approx_sin) and refer to the Simulink Fixed Point documentation.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 10. Use reusable subsystems

A single reusable subsystem can replace groups of common blocks. In the generated code, the reusable subsystem becomes a reusable function. Only the calling arguments change. This decreases ROM consumption. (You can also reuse an entire model, which will reduce ROM). For example, the equation subsystem is replicated in the model below as Equation1.

```
>> ReusableSubsystem
```

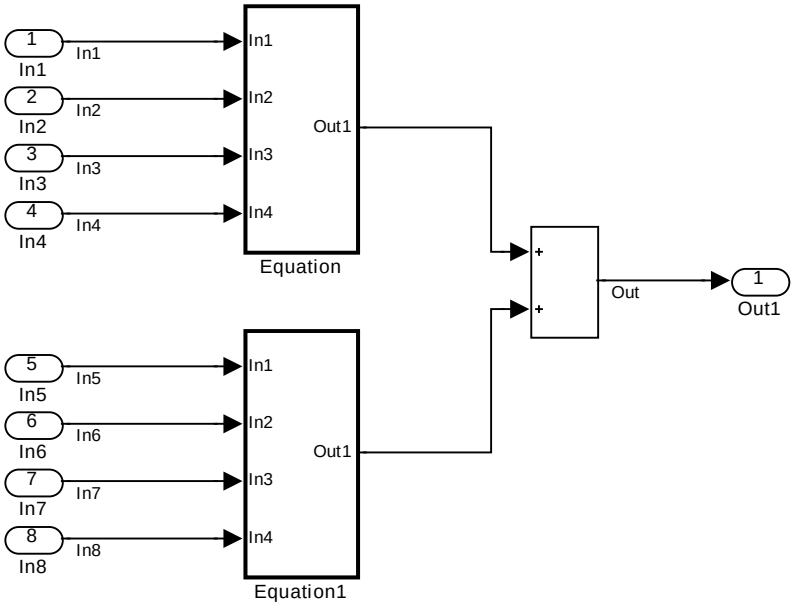


Figure 39: Model ReusableSubsystem.mdl.

Selecting the **Treat as atomic unit** check box, and selecting **Reusable function in the RTW system code** field on the Block Parameters dialog makes the subsystem Equation reusable

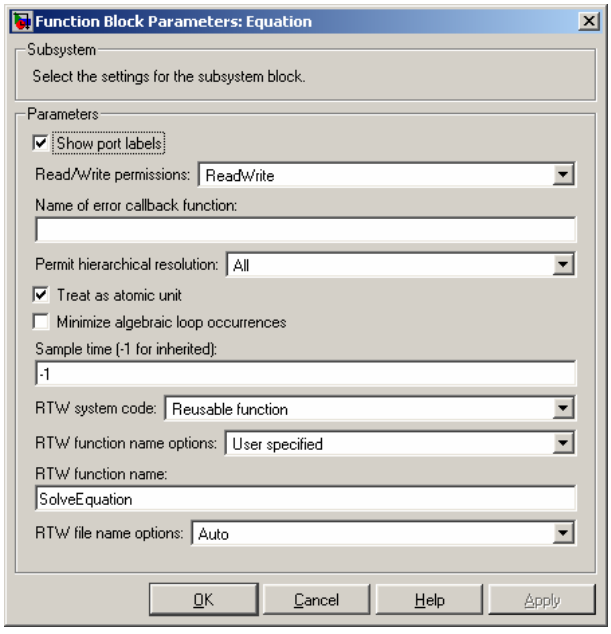


Figure 40: Settings for reusable subsystem.

© COPYRIGHT 2005 by The MathWorks, Inc.

The content of the equation subsystem is shown below.

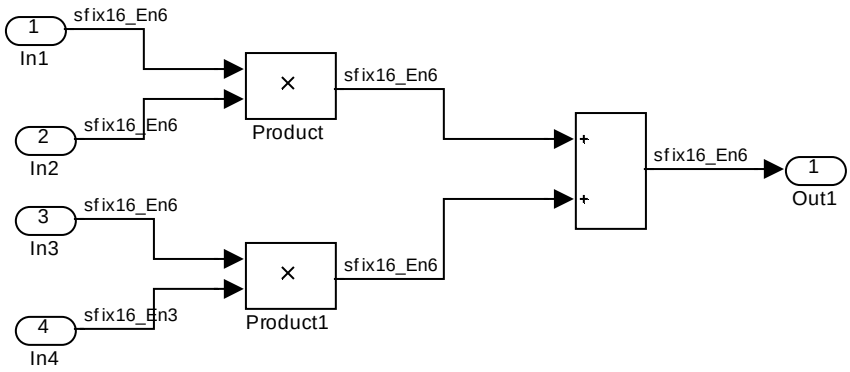


Figure 41: Equation subsystem in ReusableSubsystem.mdl.

Notice in the generated code shown below that the function `SolveEquation ()` is invoked twice, with appropriate arguments for each instance.

```
int16_T Out;

BlockIO_ReusableSubsystem ReusableSubsystem_B;

void SolveEquation(int16_T rtu_0, int16_T rtu_1, int16_T rtu_2, int16_T rtu_3,
   rtB_SolveEquation *localB)
{
   localB->Sum = (int16_T)((int32_T)rtu_0 * (int32_T)rtu_1 >> 6) +
      (int16_T)((int32_T)rtu_2 * (int32_T)rtu_3 >> 3);
}

void ReusableSubsystem_step(void)
{
   SolveEquation(In1, In2, In3, In4, &ReusableSubsystem_B.Equation);

   SolveEquation(In5, In6, In7, In8, &ReusableSubsystem_B.Equation1);

   Out = ReusableSubsystem_B.Equation.Sum + ReusableSubsystem_B.Equation1.Sum;
}
```

Figure 42: Reusable functions in the generated code.

© COPYRIGHT 2005 by The MathWorks, Inc.

To make a reusable *subsystem*,

1. Open the model.
2. Find any two or more groups of blocks whose input and output data type and scaling are identical.
3. Make one of the groups a subsystem as follows:
    a. Select all the blocks in the group, and then select **Create subsystem** on the **Edit** menu.
    b. Right-click the subsystem and select **Subsystem parameters**. The Block Parameters: Subsystem dialog appears.
    c. Select **Treat** as atomic unit and, in the RTW system code field, select **Reusable function**. Click **OK**.
    d. Select all the blocks in the second group (or groups) and delete them. This leaves an empty area with unconnected inputs and outputs.
    e. Copy the subsystem you made and paste it in the empty area or areas.
    f. Connect all the inputs and outputs to the subsystem or subsystems.
4. Generate code. The code contains a single reusable function for the subsystems.

To make a reusable *model*,

1. Open the model.
2. Select **Configuration parameters** on the Simulation menu.
3. Select **Real-Time Workshop**
4. Select **Browse** and select **System target file** as `ert.tlc`.
5. Select **Interface** subsection
6. In the **Code interface** options, **select Generate reusable code**.
7. Generate code. The code contains a single reusable function for the entire model.

Limitations on Reusability

Atomic subsystems that are superficially identical still might not generate reusable code. Specifically, Real-Time Workshop cannot reuse subsystems having any of the following characteristics:

1) Two input ports of the reusable subsystem A share a single signal source, but the reusable subsystem B is connected to two signal sources. To Simulink, subsystem A appears to have a single input while subsystem B has two inputs, so a reusable subsystem is not created.
2) Subsystem code cannot be reused if instances of the subsystem contain:
    a. Identical input signals with dissimilar sample times. This can be avoided if subsystem is sample time independent.
    b. Identical input signals with dissimilar dimensions.
    c. Identical input signals with dissimilar data types.
    d. Identical blocks (inside the subsystems) with different names - this is *only* true if the subsystem name is fixed rather than based on the block name.

3) When tunable parameters are used in reusable subsystems with different values, the subsystems must be masked to enable reusability. Note: Inline parameters must be selected to enable tunable parameters.

4) The presence of the following blocks:
    a. Scope blocks (with data logging enabled)
    b. To File blocks
    c. To Workspace blocks
    d. S-function blocks that fail to meet the following requirements:
        - The S-function must be inlined.
        - Code generated from the S-function must not use static variables.

- The TLC code that generates the inlined S-function code must not use the BlockInstanceData function.
- The S-function must initialize its pointer work vector in `mdlStart` and not before.
- The S-function must not be a sink that logs data to the workspace.
- The S-function must register its parameters as run-time parameters in `mdlSetWorkWidths`. (It must not use `ssWriteRTWParameters` in its `mdlRTW` function for this purpose.)
- In addition to meeting the preceding requirements, your S-function must set the `SS_OPTION_WORKS_WITH_CODE_REUSE` flag (see `ssSetOptions`). This flag assures Real-Time Workshop that your S-function meets the requirements for subsystem code reuse.

Some of these situations can arise even when subsystems are copied and pasted within or between models or are manually constructed to be identical. If Real-Time Workshop determines that the code for a subsystem cannot be reused, it outputs the subsystem as a function with a mangled name and arguments when the Reusable function is selected. The function is re-entrant, but it is not reused. If you prefer that subsystem code be inlined in such circumstances rather than deployed as functions, you should choose **Inline** for the **RTW system code** option. For additional information, refer to the Real-Time Workshop documentation.

Generating reusable code from Stateflow charts
You can generate reusable code from a Stateflow chart, or from a subsystem containing a Stateflow chart, except in the following cases:

- The Stateflow chart contains exported graphical functions.
- The Stateflow model contains machine parented events.

Tip 11. Use shared utility feature

A shared function (utility) can be used by blocks within the same model and by blocks in different models when using model reference or when building multiple stand-alone models from the same build directory. However, a given function is only generated once by whichever block first triggers its generation. As subsequent blocks determine the need to generate the same function, a file existence check is done and if it the file exists, a function is not generated. Thus, the shared utility function mechanism requires that a given function and filename must represent the same functional behavior regardless of which block or model generates the function.

If no model (reference) blocks are present, any code required for fixed point and other shared utilities is placed in the `model.c` file or in the build directory, not in the `slprj/<target>/_sharedutils` directory. Figure 43 shows the options available for placement of code generated for fixed-point and other shared utilities.

© COPYRIGHT 2005 by The MathWorks, Inc.

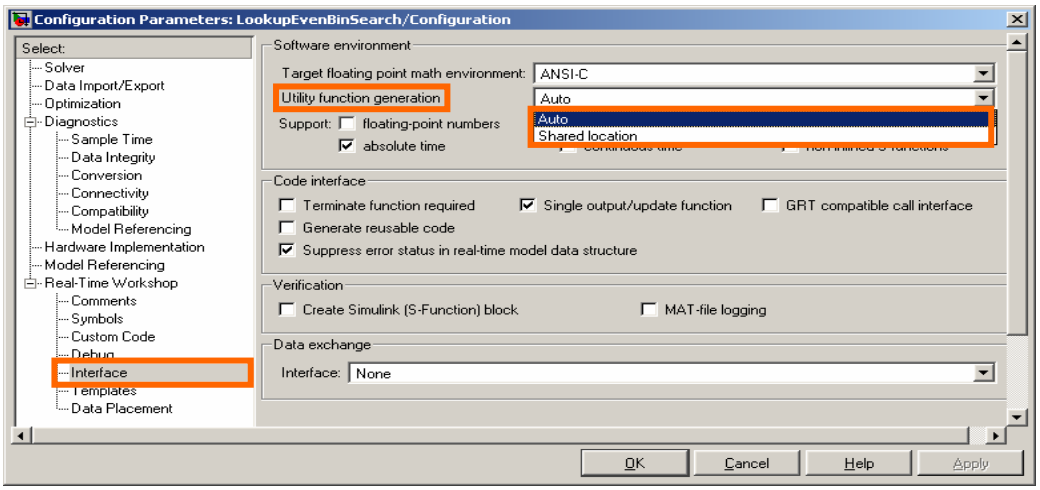


Figure 43: Settings for placement of shared utilities code

To force a build to use the *slprj</target>/_sharedutils* directory even when the current model contains no model (reference) blocks, set the Utility function generation from the drop-down list on the Real-Time Workshop Interface pane to `Shared location`. This places the utilities in the slprj directory rather than in the build directory. This is useful when manually combining code from several models as it prevents symbol collisions between models.

The other option (Auto) places utility functions in the slprj directory only when there are referenced models. That is, if a model does contain model blocks, then the `Auto` setting of the Utility function generation drop-down list uses the shared utilities directory within slprj. If a model does not contain model blocks, then the code is placed in the model.c file. If you are building multiple models from the same directory (as the utilities are placed in the model.c file) the utilities are not shared but generated for each model separately. If you are combining code from several models, this could lead to collision of symbols and increase ROM.

Shared utility feature uses the checksum mechanism to ensure that several critical properties (hardware implementation, etc.) set in the Configuration Parameters dialog are identical for all models (when building multiple models from the same directory) that are using the shared utilities. For the fixed-point functions the data type, scaling, overflow handling method, and rounding mode also must be identical for a given block to share a utility. The naming convention for the fixed-point utilities is as follows:
<operation>+[<zero protection>]+<output data type>+<output bits>+[<input1 data type>]+<input1 bits> +[<input2 data type>]+<input2 bits> + [<shift direction>]+[<saturate mode>]+[<round mode>]
Below are examples of generated fixed-point utility files, the function names in the file are identical to the file name without the extension: *FIX2FIX_U12_U16.c, FIX2FIX_S9_S9_SR9.c, MUL_S30_S30_S16.h*

To save ROM, enable the shared utility feature and build stand-alone models (without model reference blocks) from the same build directory. For additional details, refer to Real-Time Workshop Embedded Coder documentation.

```
>> shared_utils
```

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 12. Use Stateflow operator 'C'

Use the C qualifier after literal constants in Stateflow to make the code generator automatically consider the usage context of the constant. This causes the code generator to use the optimal data type for that constant.

The figure below shows a Stateflow chart in a Simulink model. The content of this Stateflow chart allows us to compare not using the C qualifier with using it. Notice that the first example, called "simple" in the figure, does not use the C qualifier with the constant 4.5. The second, called "better," does. (The "c" is not case sensitive.)

Figure 45 shows the generated code. In the first equation (simple), since the C qualifier was not specified in the Stateflow chart for the constant 4.5, the addition operation occurs in floating-point. In contrast, since the c qualifier was specified in the second equation (better), the code generator converted the 4.5 constant to its equivalent fixed-point value (36). It did so based on context. "Context" refers to the data type and scaling of the other operands in the expression. You can also explicitly specify parameter objects in the base workspace and not rely on context. In this example, the data type and scaling of the constant is that of the variable "input," namely `sfix16_En3`. Therefore, using the c qualifier automatically selects the appropriate data type and scaling of the constant, based on the context. Then the addition occurs with integer mathematics. This decreases ROM consumption and execution time compared with the first example.

```
>> SfContextC
```

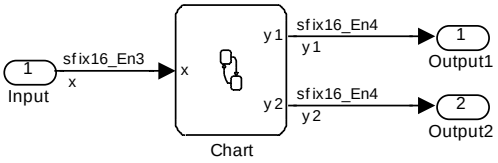


Figure 44: Model SfContextC.mdl

```
int16_T y1;

int16_T y2;


void SfContextC_step(void)
{
   /*  simple - no context sensitive literal  */
   y1 = (int16_T)ldexp(ldexp((real_T)x, -3) + 4.5, 4);

   /*  better - with context sensitive literal  */
   y2 = (x + 36) << 1;

 }
```

Figure 45: Code comparing absence and inclusion of C qualifier.

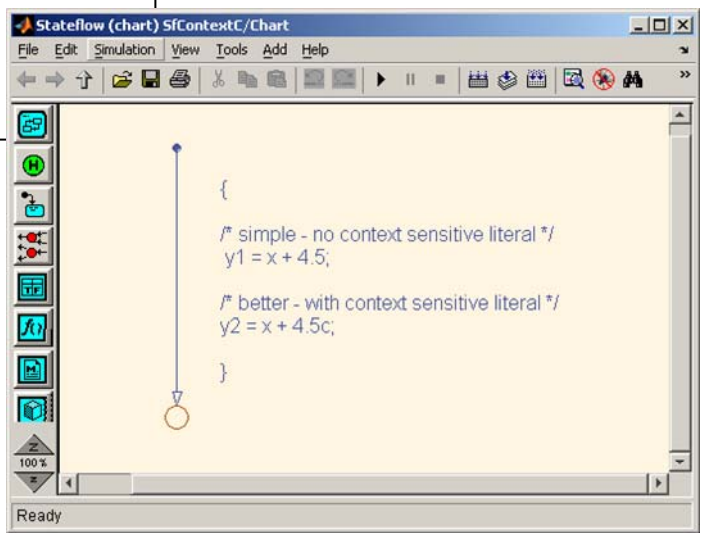

Stateflow (chart) SfContextC/Chart
File Edit Simulation View Tools Add Help
{
/* simple - no context sensitive literal */
y1 = x + 4.5;
/* better - with context sensitive literal */
y2 = x + 4.5c;
}
100 %
Ready

Figure 46: Content of Stateflow chart in SfContextC.mdl.

© COPYRIGHT 2005 by The MathWorks, Inc.

Designing for RAM reduction

The following tips can be used to design your model in order reduce the RAM consumption time required to execute the Real-Time Workshop Embedded Coder generated code from the model.

- Tip # 13: Use base data types to conserve RAM (RAM)
- Tip # 14: Limit the use of ”non-Auto” storage classes (RAM)

The following is a list of additional optimizations that enable reduction in RAM. You can find details in the Real-Time Workshop Embedded Coder on these optimizations:

- Local block outputs
- Parameter inlining (in order to reduce the use of globals)

The following examples, which also help reduce RAM consumption, were discussed in the preceding sections:

- Limit the use of unevenly spaced lookup tables (Tip #7). When the lookup table data is evenly spaced, the (input axis) x-axis data in not required. Therefore, there is no data generated for this, conserving RAM.
- Limit use of reusable subsystems for simple/small subsystems (Tip #10). In other words, if your goal is to conserve RAM over ROM and execution time, then do not implement Tip #10 in your model.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 13. Use base data types to conserve RAM

Using the fixed-point data type word lengths so that they are equal to or less than your target controller will save RAM.

For example, as shown in Figure 47, Unit Delay blocks have discrete states that have the same word lengths as their I/O signals. These discrete states are global variables that consume RAM. In addition, as shown in Figure 48, the code generated for a model containing base data types is much simpler and consumes less stack space (needed for locals).

```
>> base_data_types_RAM
```

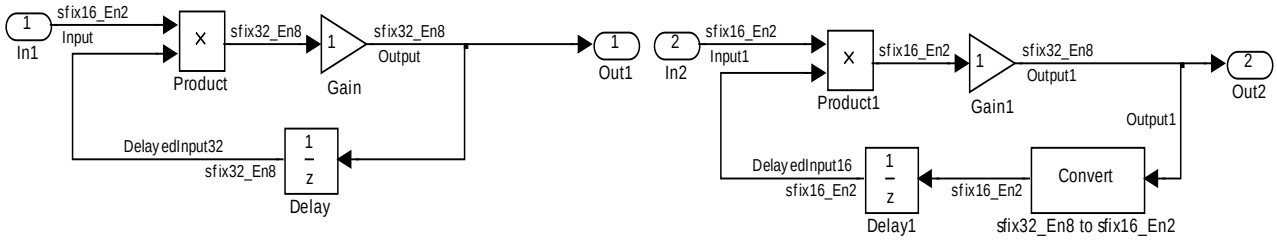


Figure 47: Model base_data_types_RAM.mdl

```
int32_T Delay;
int16_T Delay1;

void base_data_types_RAM_step(int_T tid)
{
    int16_T rtb_sfix32_En8tosfix16_En2;
    {
      int32_T _fixptlowering0;

      Output = mul_s32_s32_s32_sr2_zero((int32_T)Input, Delay);

      _fixptlowering0 = (int32_T)Input1 * (int32_T)Delay1;
      Output1 = (int32_T)(int16_T)((_fixptlowering0 >> 2) +
       (int32_T)((_fixptlowering0 < 0L) && (_fixptlowering0 & 3L))) << 6;

      rtb_sfix32_En8tosfix16_En2 = (int16_T)(Output1 >> 6);

      Delay  = Output;

      Delay1 = rtb_sfix32_En8tosfix16_En2;
    }
}
```

The code for delay block with base data type (16bits) is simpler (doesn't call mul_s32_s32_s32_sr2_zero)and uses less RAM

Figure 48: Use base data types to conserve RAM

If possible, restrict the fixed-point data type word lengths to the size of your target controller.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 14. Limit the use of non-auto storage classes

To obtain more benefits of expression folding, in addition to those mentioned in Tip #5, set the *RTW storage class* field on the Signal Properties dialog to Auto for each signal. This is the default. If you choose a setting in *RTW storage class* other than Auto, a separate statement will be generated. However, some signals need the storage class setting to be other than Auto. Change the storage class to a selection other than Auto only for those signals that need visibility (i.e., need to be global variables). For example, signals for calibration need to be global variables. Otherwise, the code may consume additional memory unnecessarily.

For the example model below, Figure 51 shows code when Auto is selected. However, when the storage class of the Gain block named MultiOutput is set to ExportedGlobal, this generates the code shown in Figure 54. In this figure, the generated code has multiplication and addition as separate statements.

```
>> ExpressionFolding
```

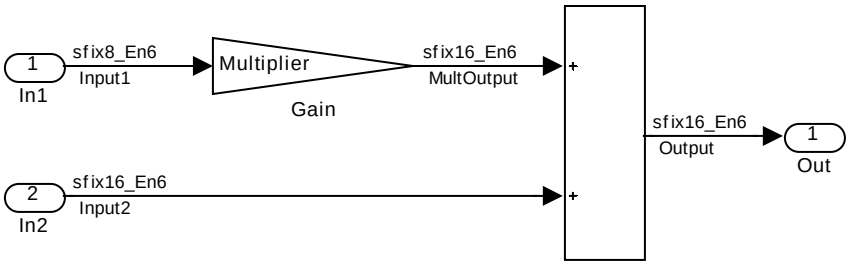


Figure 49: Model ExpressionFolding.mdl.

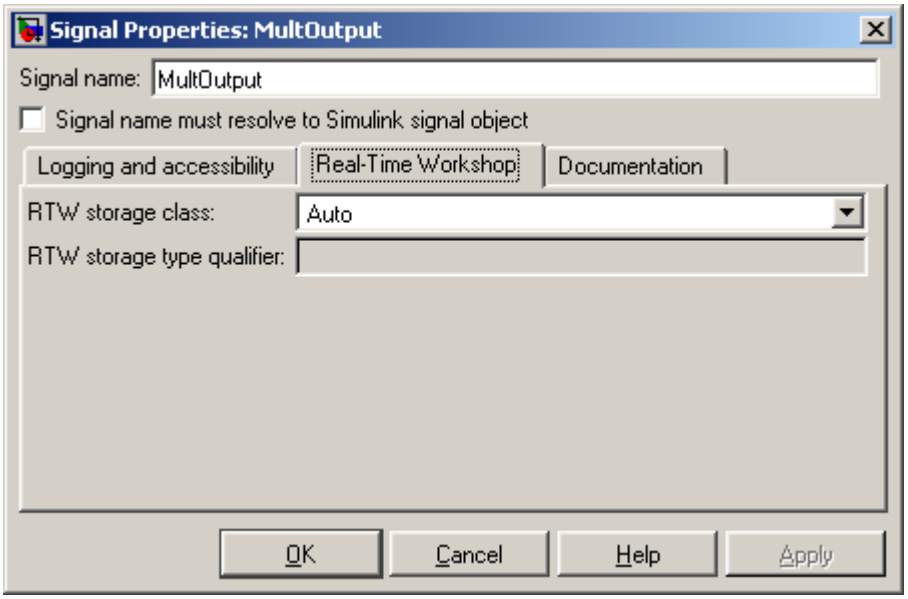


Figure 50: Setting that minimizes benefit of expression folding.

© COPYRIGHT 2005 by The MathWorks, Inc.

```
int16_T Output;

const int8_T Multiplier = 77;

void ExpressionFolding_step(void)
{

   Output = (Input1 * Multiplier >> 6) + Input2;
}
```

Figure 51: Code with auto selection.

```
>> ExpressionFoldingSigName
```

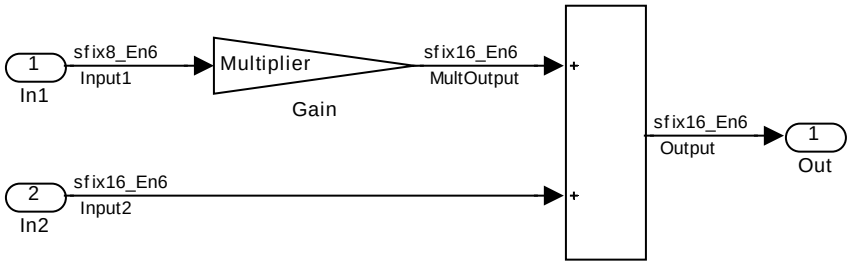


Figure 52: Model ExpressionFoldingSigName.mdl.

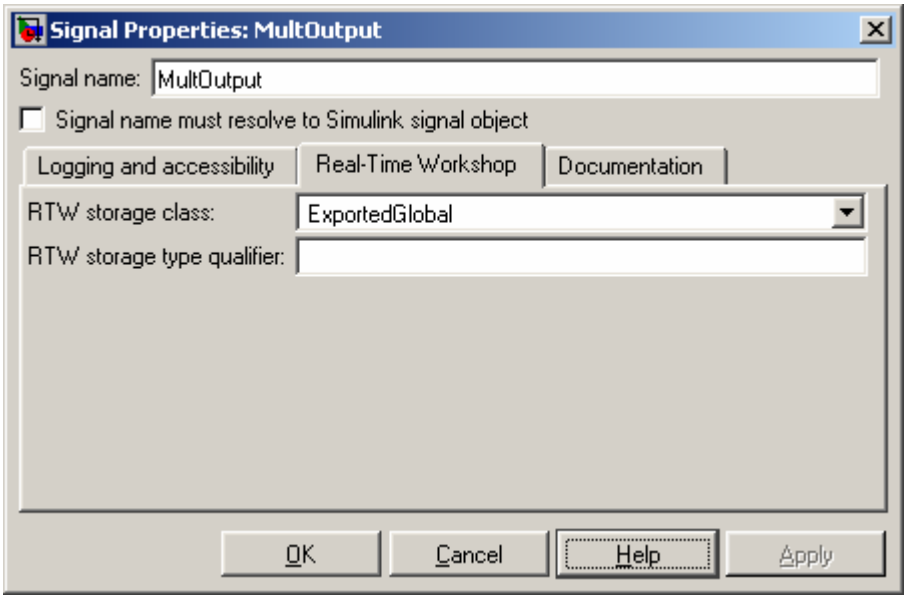


Figure 53: Code with non-auto (ExportedGlobal) selection.

© COPYRIGHT 2005 by The MathWorks, Inc.

```
int16_T MultOutput

int16_T Output;

const int8_T Multiplier = 77;

void ExpressionFoldingSigName_step(void)
{
   MultOutput = Input1 * Multiplier >> 6;

   Output = MultOutput + Input2;
}
```

Figure 54: Code with non-auto (ExportedGlobal) selection

For all of the signals in the model that do not need visibility:

1. Right-click the signal. The Signal Properties dialog appears.
2. Select **Real-Time Workshop** tab and ensure that `Auto` is selected in the **RTW storage class** field or see Tip 30 for an easier way.

For all of the signals in the model that need visibility:

1. Right-click the signal. The Signal Properties dialog appears.
2. Select (use bold again here) tab and choose the appropriate selection in the **RTW storage class** field other than `Auto`, or see Tip 30 for an easier way.
3. Generate code.

Alternatively, objects can be created by the data object wizard to control the visibility (storage class). Refer to Real-Time Workshop documentation for additional information.

© COPYRIGHT 2005 by The MathWorks, Inc.

Designing for accuracy

The goal of preserving accuracy is generally contrary to preserving RAM, ROM and execution time. To improve or preserve accuracy, a particular design may utilize more ROM, RAM and execution time.

The following tips can be used in designing your model to preserve the accuracy of the results of fixed-point operations of the Real-Time Workshop Embedded Coder generated code from your model.

- Tip 15. Use data types larger than base data types
- Tip 16. Avoid multiple multiply and divide operations
- Tip 17. Minimize internal conversion issues
- Tip 18. Use Stateflow operators ”C” or “c” and ‘:=‘
- Tip 19. Use Power of two spaced breakpoints in lookup tables (see Tip # 9)

Tip 15. Use base data types larger than base data types

As discussed in the section Fixed Point vs. Floating Point, fixed-point data types, due to smaller dynamic, are more prone to quantization errors. Larger range and higher precision can be accommodated when larger data types are used.

To improve accuracy, use data types larger than base types.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 16. Avoid multiple multiply or divide operations

The order in which multiplications and divisions are arranged in a model can have a big impact on accuracy and efficiency.

Case 1:
When a product block is configured to perform more than one division operation, it can result in a loss of accuracy and execution speed. A general guideline is to multiply all the denominator terms together first, then perform one and only one division. This improves accuracy and often execution time of fixed-point operations. This can be accomplished in Simulink by cascading product blocks.

Case 2:
When a single product block is configured to perform more than one multiplication or division operation, it is difficult to control the data types for intermediate results. Therefore, it can result in a loss of accuracy.

If the output data type is integer or fixed-point, then better results are likely if this operation is split across several blocks each doing one multiplication or one division. Using several blocks allows the user to control the data type and scaling used for intermediate calculations. The choice of data types for intermediate calculations affects precision, range errors, and efficiency.

```
>> multiple_mul_div
```

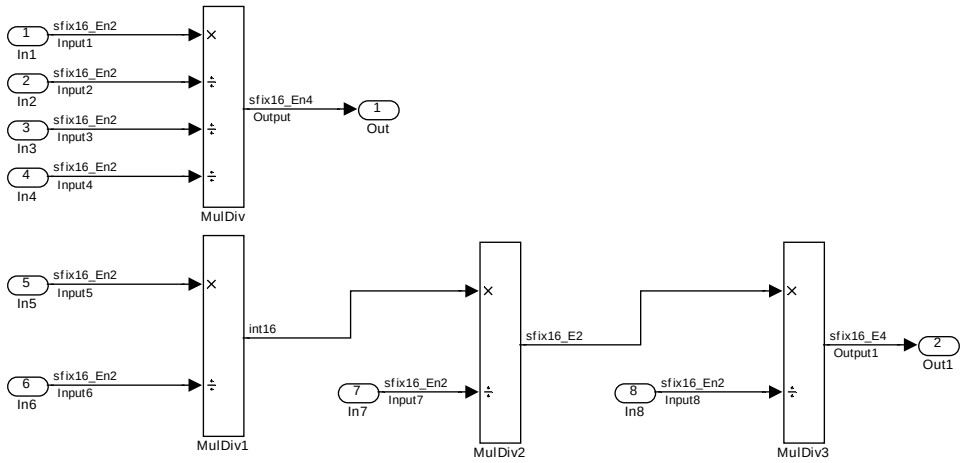


Figure 55: Model multiple_mul_div.mdl

```
void multiple_mul_div_step(void)
{

     Output =
        div_s16s32_floor((int32_T)div_s16s32_floor((int32_T)div_s16s32_floor(
        (int32_T)Input1 << 4, (int32_T)Input2) << 2, (int32_T)Input3) << 2,
        (int32_T)Input4);

     Output1 = div_s16_floor(div_s16_floor(div_s16_floor(Input5, Input6), Input7),
      Input8);

}
```

Figure 56: Code for multiple operations in one block versus one operation per block

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 17. Minimize internal conversion issues

Fixed-point operations are based on lower level operations, such as integer addition and comparisons, which require the inputs to have the same data type and scaling. If not, an internal conversion is needed. Blocks built on these operations are Relational Operator, MinMax, and Sum. For some operations, the need to do an internal conversion can represent a design oversight. The impact of this oversight is loss of accuracy (due to overflows) and efficiency (execution speed).

Minimize internal conversions by matching data type and scaling for Relational Operator, MinMax, and Sum blocks.

```
>> internal_conv_issues
```

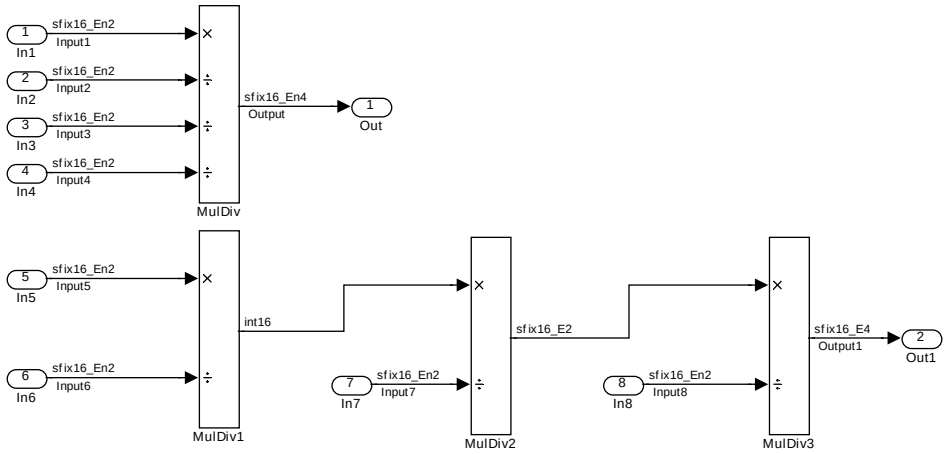


Figure 57: Model internal_conv_issues.mdl

```
void internal_conv_issues_step(void)
{
      int32_T rtb_MinMax;
      int16_T rtb_Sum;
      boolean_T rtb_Output;

      {
        int32_T castIn1;

        castIn1 = FIX2FIX_S24_S16_SR1(Input3);
        if ( Input2 > castIn1 )
        {
          rtb_MinMax = castIn1;
        }
        else
        {
           rtb_MinMax = Input2;
        }
       }

       rtb_Sum = ((int16_T)rtb_MinMax);
       rtb_Sum += ASR(2,Input4);

       rtb_Output = (Input1) <= (FIX2FIX_S24_S16_SL1(rtb_Sum));

       internal_conv_issues_Y.Out1 = rtb_Output;

       internal_conv_issues_Y.Out2 = (internal_conv_issues_U.In5 <=
         rt_MIN(internal_conv_issues_U.In6,
         internal_conv_issues_U.In7) + internal_conv_issues_U.In8);

}
```

Figure 58: Code for internal_conv_issues

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 18. Use Stateflow operators ‘C’ or ‘c” and ‘:=’

There are two special features in Stateflow for performing fixed-point mathematical operations, namely the “:=” operator and the “c” (or “C”) qualifier. The ‘:=’ operator yields results that are more precise. The c qualifier automatically selects the appropriate data type and scaling of a constant, based on context. Using these will produce a higher degree of optimized embedded code. Here is an explanation of each. (See the Stateflow documentation for additional details.)

The := assignment operator

Using the := operator in a Stateflow chart instead of the = operator is useful, especially in multiplication and division. The := preserves precision in the result of multiplication or division that = may not retain. The := is less useful in addition and subtraction as to precision. Nevertheless, even here its use can avoid overflow. The figure below shows a Simulink model in which there is a Stateflow chart that has two inputs and outputs. The content of the Stateflow chart, shown in the next figure, allows us to compare the use of = and :=. The “general” case performs multiplication and division using =. The “better” case performs the same multiplication and division using := instead.

Figure 61 shows the generated code. In the multiplication example, the casting of this 32-bit intermediate result (`int16_T`) occurs before the shifting (“`>>`”), in the general case. This can result in losing useful bits. However, in the better case, the casting takes place after the shifting. This preserves bits, yielding a more accurate result.

Now we will compare the division. In the general case, all division occurs with 16-bit operands, whereas in the := case all division occurs with 32-bit operands. The example using the := operator yields a more accurate result.

Therefore, using := in mathematical operations yields more precise results. However, note that := may increase ROM consumption and execution time compared with using the = operator.

```
>> SfColonEqual
```

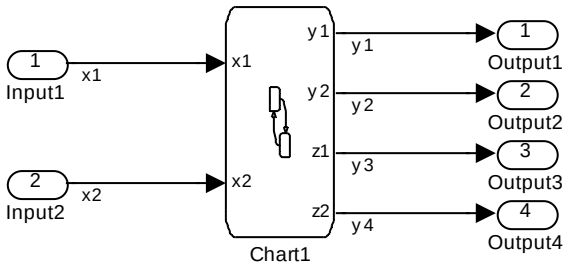


Figure 59: Model SfColonEqual.mdl.

© COPYRIGHT 2005 by The MathWorks, Inc.

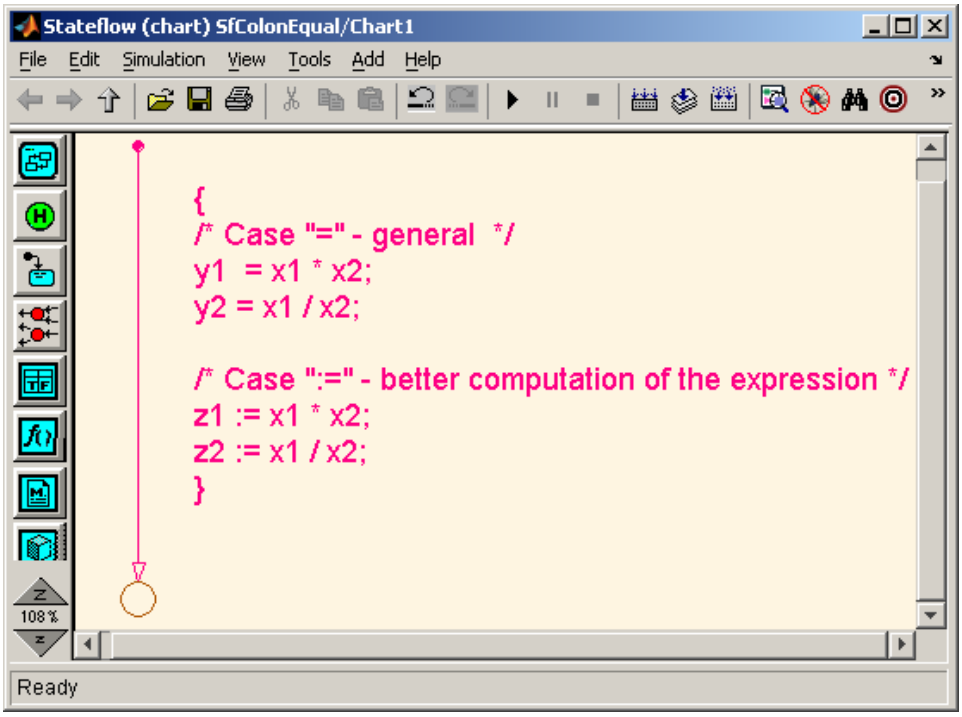

Stateflow (chart) SfColonEqual/Chart1
File Edit Simulation View Tools Add Help
{
/* Case "=" - general */
y1 = x1 * x2;
y2 = x1 / x2;
/* Case ":=" - better computation of the expression */
z1 := x1 * x2;
z2 := x1 / x2;
}
108%
Ready

Figure 60: Content of Stateflow chart in SfColonEqual.mdl.

```
int16_T y1;
int16_T y2;
int16_T y3;
int16_T y4;

void SfColonEqual_step(void)
{
     /*  Case "=" - general   */
   y1 = (int16_T)((int32_T)x1 * (int32_T)x2) >> 3;
   y2 = x1 / x2 << 3;

     /*  Case ":=" - better computation of the expression  */
   y3 = (int16_T)((int32_T)x1 * (int32_T)x2 >> 3);
   y4 = (int16_T)(((int32_T)x1 << 3) / (int32_T)x2);

}
```

Figure 61: Code comparing the = and := assignment operators.

Using the C qualifier

The use of C qualifier was also discussed earlier in Tip 12. The figure below shows a Stateflow chart in a Simulink model. The content of this Stateflow chart allows us to compare not using the C qualifier with using it. Notice that the first example, called “simple” in the figure, does not use the C qualifier with the constant 4.5. The second, called “better,” does. (The “c” is not case sensitive.)

© COPYRIGHT 2005 by The MathWorks, Inc.

Figure 55 shows the generated code. In the first equation (simple), since the C qualifier was not specified in the Stateflow chart for the constant 4.5, the addition operation occurs in floating-point. In contrast, since the C qualifier was specified in the second equation (better), the code generator converted the 4.5 constant to its equivalent fixed-point value (36). It did so based on context. "Context" refers to the data type and scaling of the other operands in the expression. In this example, the data type and scaling of the constant is that of the variable "input," namely `sfix16_En3`. Therefore, using the C qualifier automatically selects the appropriate data type and scaling of the constant, based on the context. Then the addition occurs with integer mathematics. This decreases ROM consumption and execution time compared with the first example.

```
>> SfContextC
```

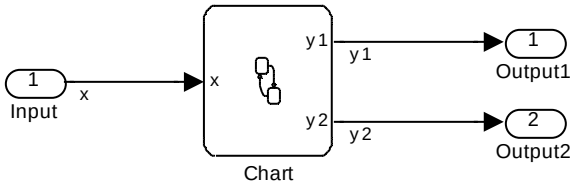


Figure 62: Model SfContextC.mdl

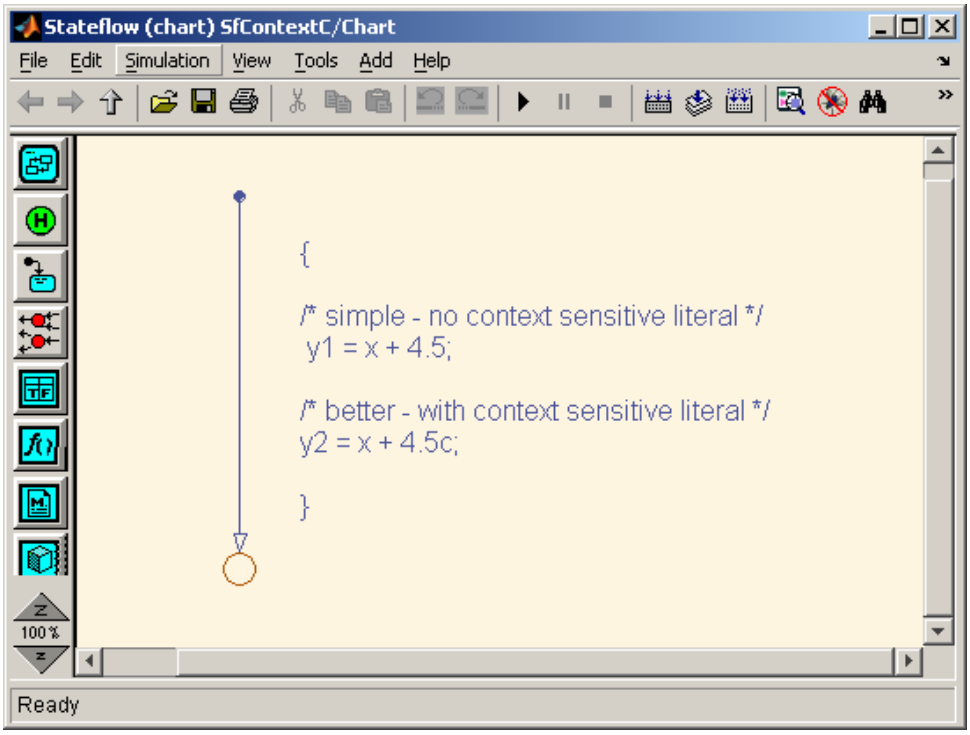

Stateflow (chart) SfContextC/Chart
File Edit Simulation View Tools Add Help
{
/* simple - no context sensitive literal */
y1 = x + 4.5;
/* better - with context sensitive literal */
y2 = x + 4.5c;
}
100 %
Ready

Figure 63: Content of Stateflow chart in SfContextC.mdl.

© COPYRIGHT 2005 by The MathWorks, Inc.

```
int16_T y1;

int16_T y2;


void SfContextC_step(void)
{

  /*  simple - no context sensitive literal  */
  y1 = (int16_T)ldexp(ldexp((real_T)x, -3) + 4.5, 4);

  /*  better - with context sensitive literal  */
  y2 = (x + 36) << 1;

}
```

Figure 64: Code comparing absence and inclusion of C qualifier.

To perform this task:

1. Replace the assignment operator = with the assignment operator := to produce generated code that is optimized for accuracy.
2. Use the C qualifier after literal constants in Stateflow to make the code generator automatically consider the usage context of the constant. Then, the code generator will use the optimal data type for that constant.

Tip 19: Use power of two spaced breakpoints in lookup tables

This tip was discussed earlier to demonstrate how to achieve reduction of ROM and execution time by using power of two spaced breakpoints in lookup tables. In addition to reducing ROM and execution time, you can also achieve accuracy (no loss of precision) with fixed-point operations. In contrast, the uneven and even cases usually introduce rounding error.

Refer to Tip #9 for details.

© COPYRIGHT 2005 by The MathWorks, Inc.

Designing for Interface

The following tips can be used to design for interfacing with software libraries, hardware drivers or other legacy code.

- Tip 20. Interface with existing variables
- Tip 21. Interface with existing typedefs or create your own
- Tip 22. Interface with existing function calls
- Tip 23. Interface legacy code with Simulink fixed-point signals

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 20. Interface with existing variables

Often times, you are required to integrate legacy code with the generated code. In certain cases, the data calculated in the legacy code needs to be utilized by an algorithm in the generated code. In the case shown in Figure 66, data at the algorithm inputs is defined in an I/O structure by default. This default behavior can be changed, as shown in Figure 68, so that data can be mapped to a name that directly references an existing variable in the legacy code.

Consider the Signal Properties dialog that appears when you right-click a signal line in a model, as shown below.

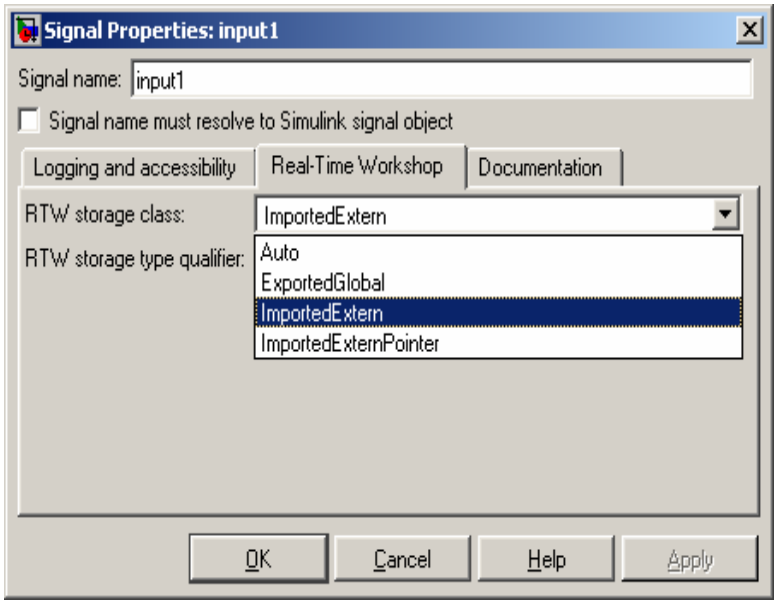


Figure 65: Signal properties dialog.

When the default `Auto` is selected in the **RTW storage class** field, Real-Time Workshop Embedded Coder generates code for all inputs into a structure called ExternalInput, and all outputs into a structure called ExternalOutput.

```
>> EliminatingIO_Struct
```

For the model below, the generated code is shown in Figure 67.

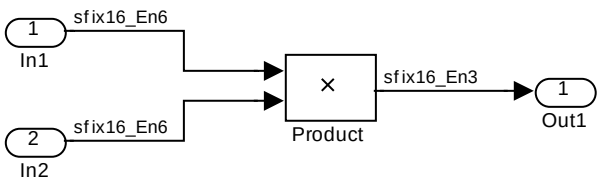


Figure 66: Model EliminatingIO_Struct.mdl

© COPYRIGHT 2005 by The MathWorks, Inc.

```
typedef struct _ExternalInputs_EliminatingIO_Struct_tag {
  int16_T In1;
  int16_T In2;
} ExternalInputs_EliminatingIO_Struct;

typedef struct _ExternalOutputs_EliminatingIO_Struct_tag {
  int16_T Out1;
} ExternalOutputs_EliminatingIO_Struct;
.
.
.

ExternalInputs_EliminatingIO_Struct EliminatingIO_Struct_U;

ExternalOutputs_EliminatingIO_Struct EliminatingIO_Struct_Y;

void EliminatingIO_Struct_step(void)
{
  EliminatingIO_Struct_Y.Out1 =
      (int16_T)((int32_T)EliminatingIO_Struct_U.In1 *
      (int32_T)EliminatingIO_Struct_U.In2 >> 9);
}
```

Figure 67: Code with ExternalInput and ExternalOutput structures.

If you choose one of the other selections in the **RTW storage class** field instead of the default `Auto`, the ExternalInput and ExternalOutput structures will not appear in the generated code. In addition, the signal names themselves will replace the generated code's "`structure_name.signal_name`" nomenclature. Furthermore, for certain compilers you may achieve the added benefit of ROM savings. This is because the code accesses the signal directly rather than through a structure.

```
>> EliminatingIO_StructSigName
```

For the model below, the **RTW storage class** was set to ImportedExtern for Input1 and Input2 and ExportedGlobal for Output . The generated code is shown in Figure 69. Compare the two figures and notice that the code in Figure 69 does not have references to structures.

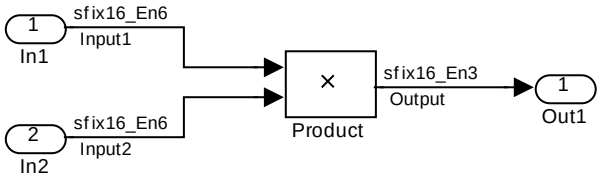


Figure 68: Model EliminationgIO_StructSigName.mdl.

© COPYRIGHT 2005 by The MathWorks, Inc.

```
extern int16_T Input1;
extern int16_T Input2;
.
.
.
int16_T Output;

void EliminatingIO_Struct_step(void)
{

  Output = (int16_T)((int32_T)Input1 * (int32_T)Input2 >> 9);
}
```

Figure 69: Code without ExternalInput and ExternalOutput structures.

Note: See the Real-Time Workshop Embedded Coder documentation for more details on storage classes.

For each signal,

1. Right-click to display the Signal Properties dialog.
2. Select **Real-Time Workshop** tab and then choose the appropriate **RTW storage class** setting (not `Auto`) or see Tip 30 for an easier way.
3. Generate code.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 21. Interface with existing typedefs or create your own

A `Simulink.NumericType` object is used to create a typedef or use an existing one. In the Model Explorer, add a new `Simulink.NumericType` object and set its values as shown in the Figure 70 to create your own typedef. For example, typedef int16_T SpeedType1. To include a header file that contains the typedef for SpeedType1, set its values as shown in Figure 71.

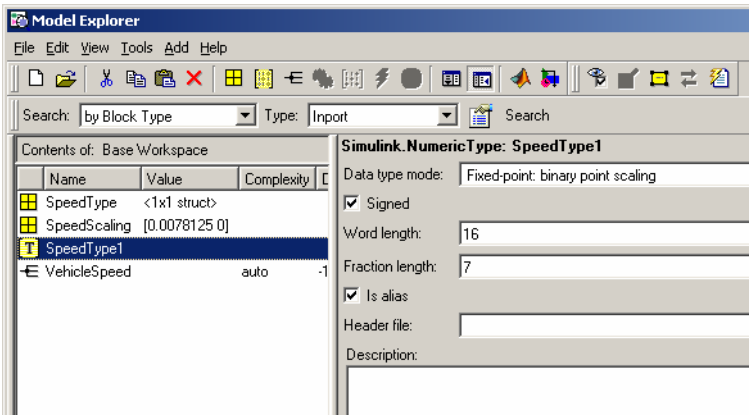


Figure 70: Create your own typedef

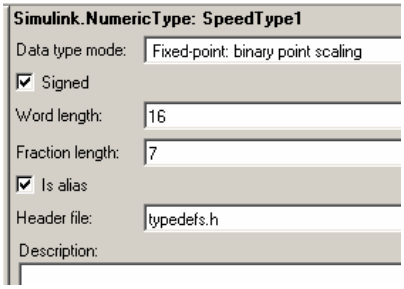


Figure 71: Include a header file that contains the typedef

```
>> numeric_types
```

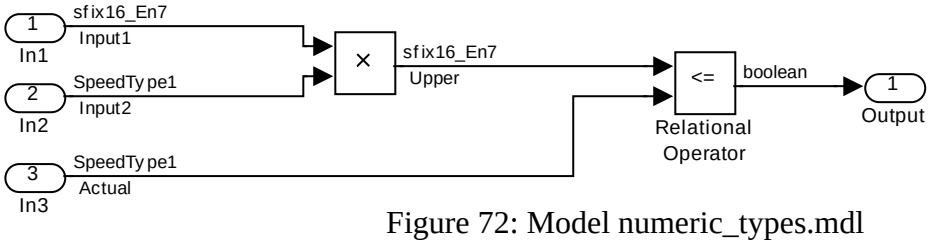


Figure 72: Model numeric_types.mdl

To use the created typedefs in the model, the output data type of block can reference the typedef as shown in Figure 73.

© COPYRIGHT 2005 by The MathWorks, Inc.

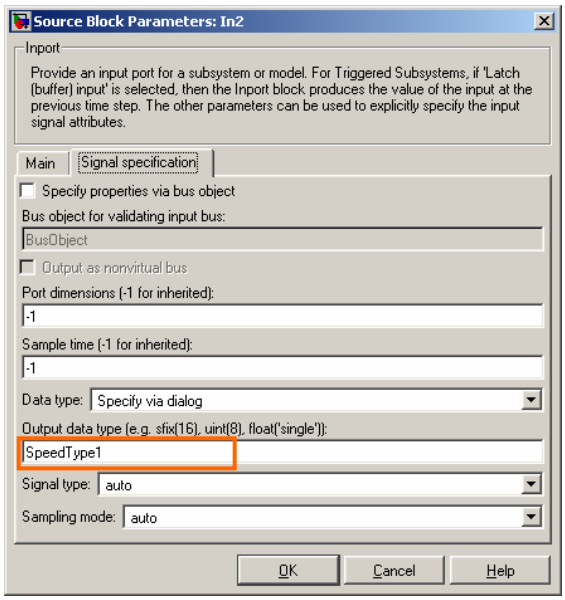

Source Block Parameters: In2
Inport
Provide an input port for a subsystem or model. For Triggered Subsystems, if 'Latch (buffer) input' is selected, then the Inport block produces the value of the input at the previous time step. The other parameters can be used to explicitly specify the input signal attributes.
Main
Signal specification
Specify properties via bus object
Bus object for validating input bus:
BusObject
Output as nonvirtual bus
Port dimensions (-1 for inherited):
-1
Sample time (-1 for inherited):
-1
Data type: Specify via dialog
Output data type (e.g. sfix(16), uint(8), float('single')):
SpeedType1
Signal type: auto
Sampling mode: auto
OK
Cancel
Help

Figure 73: Set output data type

In addition to creating a typedef and using an existing typedef, Simulink.NumericType objects can be used to easily and consistently (eliminate errors) specify scaling of signals with same data type and scaling.

Instead of using a Simulink.NumericType, you can use MATLAB workspace variables to specify the Output data type. However, this method will not yield a typedef and you will have to specify **Output data type** and **Output scaling value** separately as shown in Figure 74 instead of just the *Output data type* as shown in Figure 73.

```
>> SpeedType = sfix(16);
>> SpeedScaling = [2^-7 0];
```

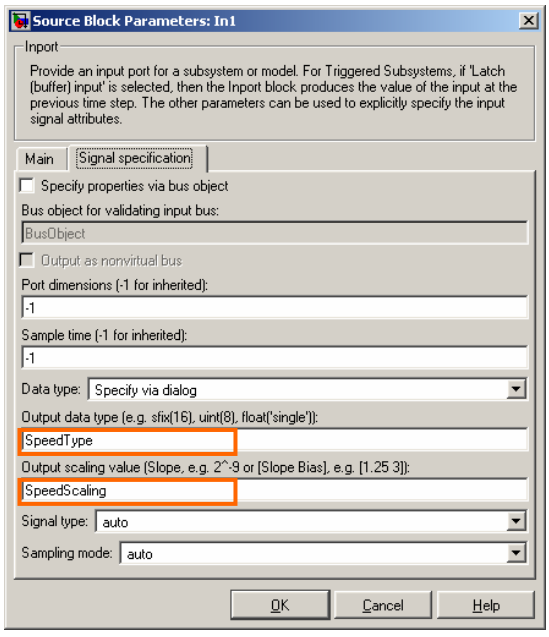

Source Block Parameters: In1
Inport
Provide an input port for a subsystem or model. For Triggered Subsystems, if 'Latch (buffer) input' is selected, then the Inport block produces the value of the input at the previous time step. The other parameters can be used to explicitly specify the input signal attributes.
Main
Signal specification
Specify properties via bus object
Bus object for validating input bus:
BusObject
Output as nonvirtual bus
Port dimensions (-1 for inherited):
-1
Sample time (-1 for inherited):
-1
Data type: Specify via dialog
Output data type (e.g. sfix(16), uint(8), float('single')):
SpeedType
Output scaling value (Slope, e.g. 2^-9 or [Slope Bias], e.g. [1.25 3]):
SpeedScaling
Signal type: auto
Sampling mode: auto
OK
Cancel
Help

Figure 74: Set output data type using MATLAB workspace variables

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 22. Interface with existing function calls

To interface with existing function calls make a subsystem non-virtual. Real-Time Workshop allows you to control how code is generated for non-virtual subsystems. The categories of nonvirtual subsystems are

- **Conditionally executed subsystems**: execution depends upon a control signal or control block. These include triggered subsystems, enabled subsystems, action and iterator subsystems, subsystems that are both triggered and enabled, and function call subsystems.

- **Atomic subsystems**: Any virtual subsystem can be declared atomic (and therefore non-virtual) via the **Treat as atomic unit** option in the Block Parameters dialog.

You can control the code generated from non-virtual subsystems as follows:

- Generate separate functions, within separate code files
- Control both the names of the functions and of the code files generated
- Cause multiple instances of a subsystem to generate reusable code, that is, as a single reentrant function, instead of replicating the code for each instance of a subsystem or each time it is called.
- Generate in-lined code

See Tip #10 for additional details.

```
>> ReusableSubsystem
```

Tip 23. Interface legacy code with Simulink fixed-point signals

To interface directly with Simulink fixed-point signals

- Use fixed-point S-functions
- Use S-functions with standard integer data types and convert the signal to fixed-point within your model using a data conversion block.
- For additional details on this topic, refer to
  - Simulink Writing S-Functions
  - Simulink Fixed Point User's Guide

© COPYRIGHT 2005 by The MathWorks, Inc.

Other modeling tips

The following tips are designed to make modeling and code generation tasks easier.

- Tip 24. Use blocks that support fixed-point production code generation
- Tip 25. Easily specify scaling and data type:
  - With back propagation or internal rule
  - With numeric types to create and apply common data type and scaling (see Tip #21)
- Tip 26. Specify target microcontroller characteristics
- Tip 27. Managing buses (structures) with fixed-point data
- Tip 28. Use Model Advisor to optimize model
- Tip 29. Reduce workload using the autoscale blockset tool
- Tip 30. Change preferences of blocks in a model globally
- Tip 31. Enforce type compatibility of parameters between workspace and model

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 24. Use blocks that support fixed-point production code generation

During the initial floating-point design that will support fixed-point implementation, select floating-point blocks that support fixed point. This eliminates the need to replace floating-point blocks using a conversion script or manual methods. With Release 14, many Simulink blocks now have intrinsic fixed point support, letting you create designs in fixed-point. To simulate or generate code in fixed-point you need Simulink Fixed-Point license.

MATLAB includes a reference table that shows the data types that support simulation and code generation for each block. This table also indicates whether or not the blocks are optimal for production code generation. The Simulink Block Data Type Support table can be displayed by

- Typing the MATLAB command `showblockdatatypetable` or

- Using the Simulink block library (as a block) or

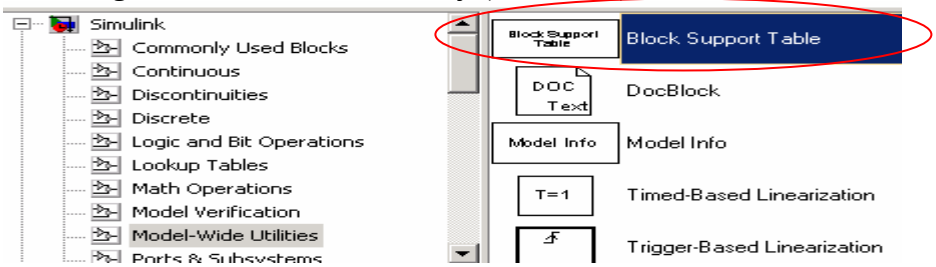


Figure 75: Use Block Support Table block from Simulink library

- Selecting **Block Support Table** from the model's Help menu

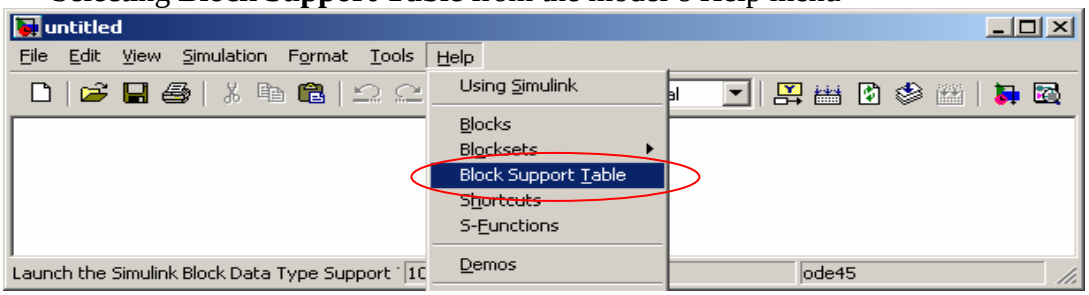


Figure 76: Select Block Support Table

A portion of this table is shown below.

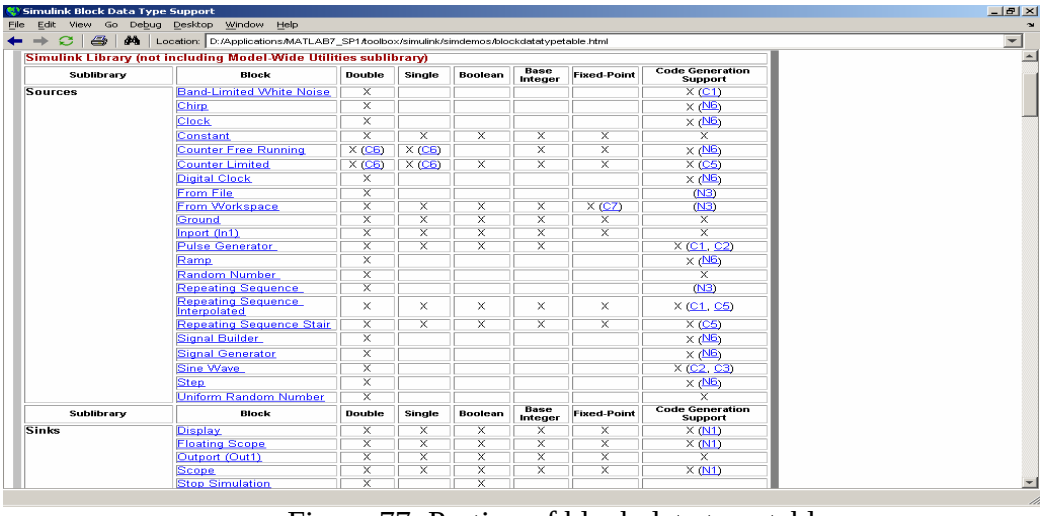


Figure 77: Portion of block data type table.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 25. Reducing time using back-propagation or internal rule for data type and scaling

For each Simulink block output, you can manually type the data type and scaling on the Block Parameters dialog. This requires:

- Double clicking the block to open the Block Parameters dialog
- Selecting the **Signal data types tab**
- Selecting `Specify via dialog` in the **Output data type mode** field, which expands the dialog more
- Typing the data type in the **Output data type** field and typing the scaling value in the **Output scaling value** field

A faster process is to select `Inherit via internal rule` or `Inherit via back propagation` in the **Output data type mode** field when appropriate, based on the algorithm. This saves you from having to manually type the data type and scaling value for the block.

When `Inherit via internal rule` is selected, Real-Time Workshop Embedded Coder selects the data type and scaling for the output that gives the best precision without possible overflows. For example, as illustrated below, multiplying the two signed eight-bit signals (`sfix8_En3 * sfix8_En2`) results in `sfix16_En5`.

```
>>InternalRule
```

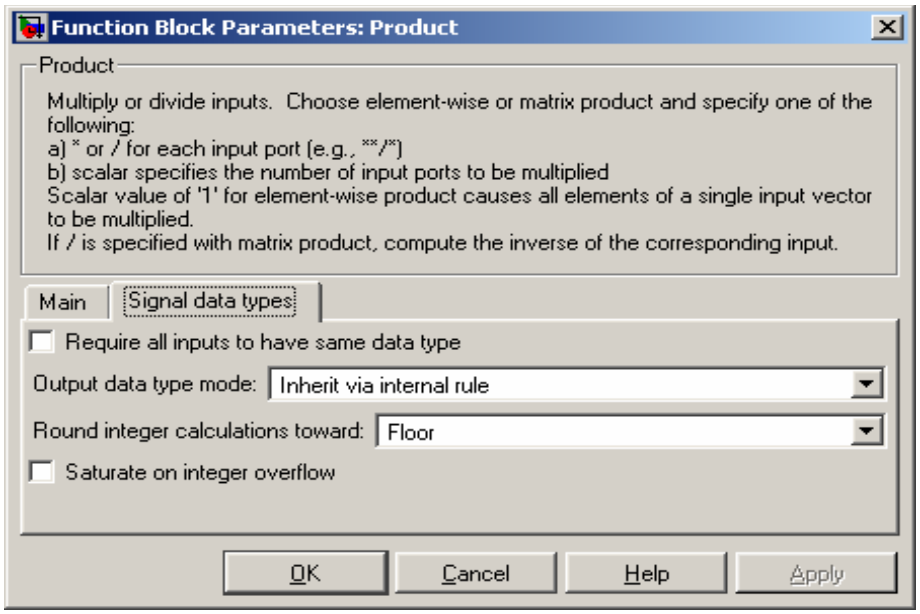

Function Block Parameters: Product
Product
Multiply or divide inputs. Choose element-wise or matrix product and specify one of the following:
a) * or / for each input port (e.g., **/*)
b) scalar specifies the number of input ports to be multiplied
Scalar value of '1' for element-wise product causes all elements of a single input vector to be multiplied.
If / is specified with matrix product, compute the inverse of the corresponding input.
Main
Signal data types
Require all inputs to have same data type
Output data type mode:
Inherit via internal rule
Round integer calculations toward:
Floor
Saturate on integer overflow
OK
Cancel
Help
Apply

Figure 78: "Inherit via internal code" setting.

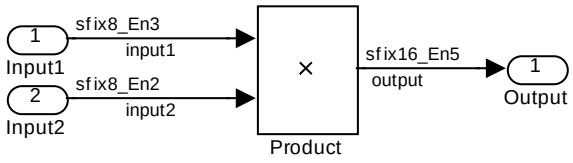


Figure 79: Model InternalRule.mdl.

© COPYRIGHT 2005 by The MathWorks, Inc.

When *Inherit via back propagation* is selected, Real-Time Workshop Embedded Coder determines the data type and scaling of the output signal by inheriting these from the block to which the output is connected. For example, as illustrated below, the Constant block output signal obtains its type (**`sfix8_En3`**) from the Relational Operator block to which the output signal is connected.

```
>> BackPropagation
```

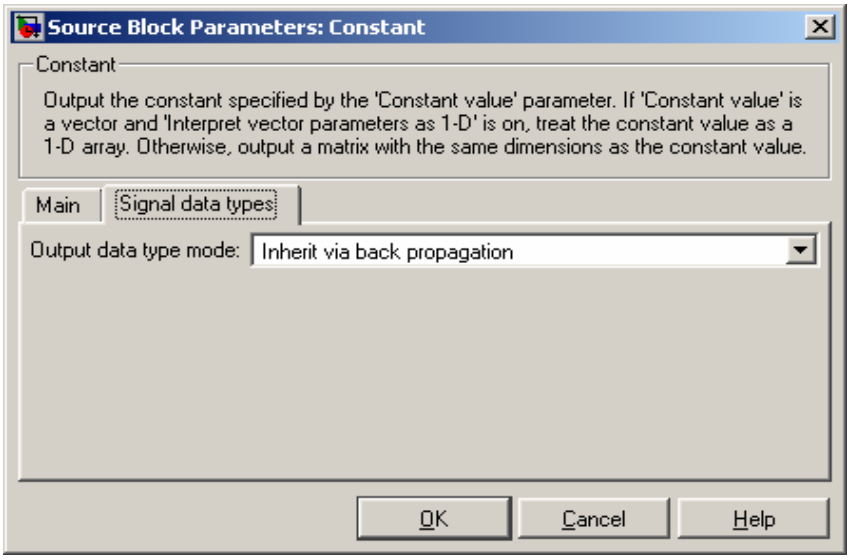

Source Block Parameters: Constant
Constant
Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.
Main
Signal data types
Output data type mode:
Inherit via back propagation
OK
Cancel
Help

Figure 80: "Inherit via back propogation" setting

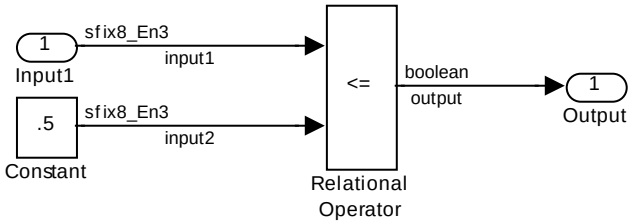


Figure 81: Model BackPropagation.mdl.

To avoid having to enter data type and scaling in both Simulink and Stateflow for each input, you can select *Inherit via back propagation* in Simulink or select *inherited* as the Type in Stateflow. For example, in Figure 82, the output from the Sum block is the input to the Limit chart. The user selects *Inherit via back propagation* in the Output data type mode field on the Block Parameters dialog for Sum, and clicks OK. Now the data type and scaling for the Sum block output (**`sfix16_En3`**) is driven solely by the Stateflow chart. Any change of data type or scaling in the Stateflow input will back propagate to Sum. Alternatively, you can select *inherited* as the Type for input in the Limit chart. Now the data type and scaling for the input to Limit chart is driven solely by the output from the Sum block. Any change of data type or scaling in Simulink output will propagate to the Stateflow chart.

© COPYRIGHT 2005 by The MathWorks, Inc.

```
>> BackPropagationSF
```

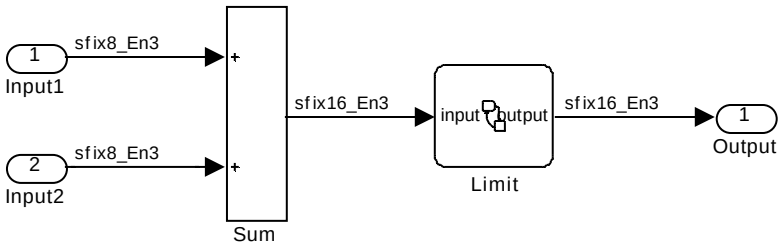


Figure 82: Model BackPropagationSF.mdl

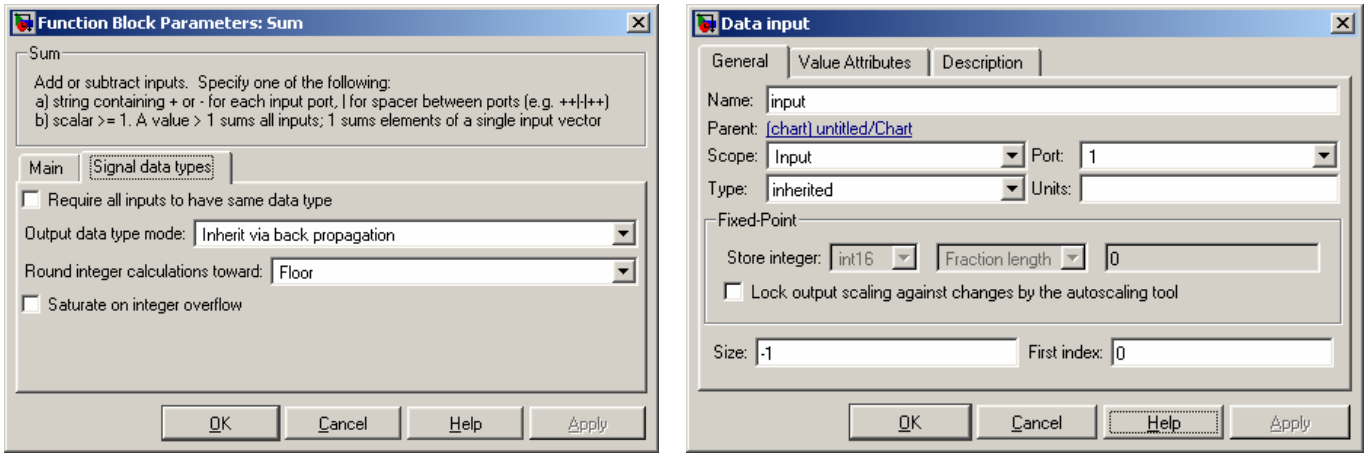


Figure 83: "Inherit for back propagation" for Sum block in Simulink or "inherited" for input in Stateflow

To avoid having to enter data type and scaling manually for each block:

1. Open the model.
2. Select either `Inherit via internal rule` or `Inherit via back propagation` in the **Output data type mode** field of the Block Parameters dialog as appropriate for each block or see Tip 30 for an easier way.
3. Generate code.

For additional information, refer to "Sharing Fixed-Point Data with Simulink" in the Stateflow documentation.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 26. Specify target microcontroller characteristics

You must specify word sizes for integer data types (for example, long, int, short) for a specific target microcontroller. The Real-Time Workshop Embedded Coder uses this information to map types in the model (for example, uint8, uint16, int8, int16) to data types that the microcontroller accommodates. If word sizes of integer data types are set inappropriately, incorrect code is generated, resulting in a compile-time error.

Using the hardware implementation configuration component, you can simultaneously specify integer and fixed-point numerical behavior for two devices:

- Embedded Hardware (simulation and code generation): The deployment hardware device for the model and the Real-Time Workshop generated code. Specifying this information in Simulink allows it to properly simulate the behavior the user can expect on eventual hardware device.

- Emulation Hardware (code generation only): The device on which code generated by Real-Time Workshop currently runs. Rapid prototyping can be carried out on hardware devices that do not match the final hardware device characteristics. The code generation process uses the prototyping hardware device characteristics in conjunction with the deployment hardware device characteristics to generate code that will behave just like it will on the deployment device.

If you are unsure of the use of these settings, it is best to set **Emulation Hardware** to `None` (the default). For additional information, refer to “Hardware Implementation Options” section in Real-Time Workshop and “Targeting an Embedded Processor” section in the Simulink Fixed Point documentation.

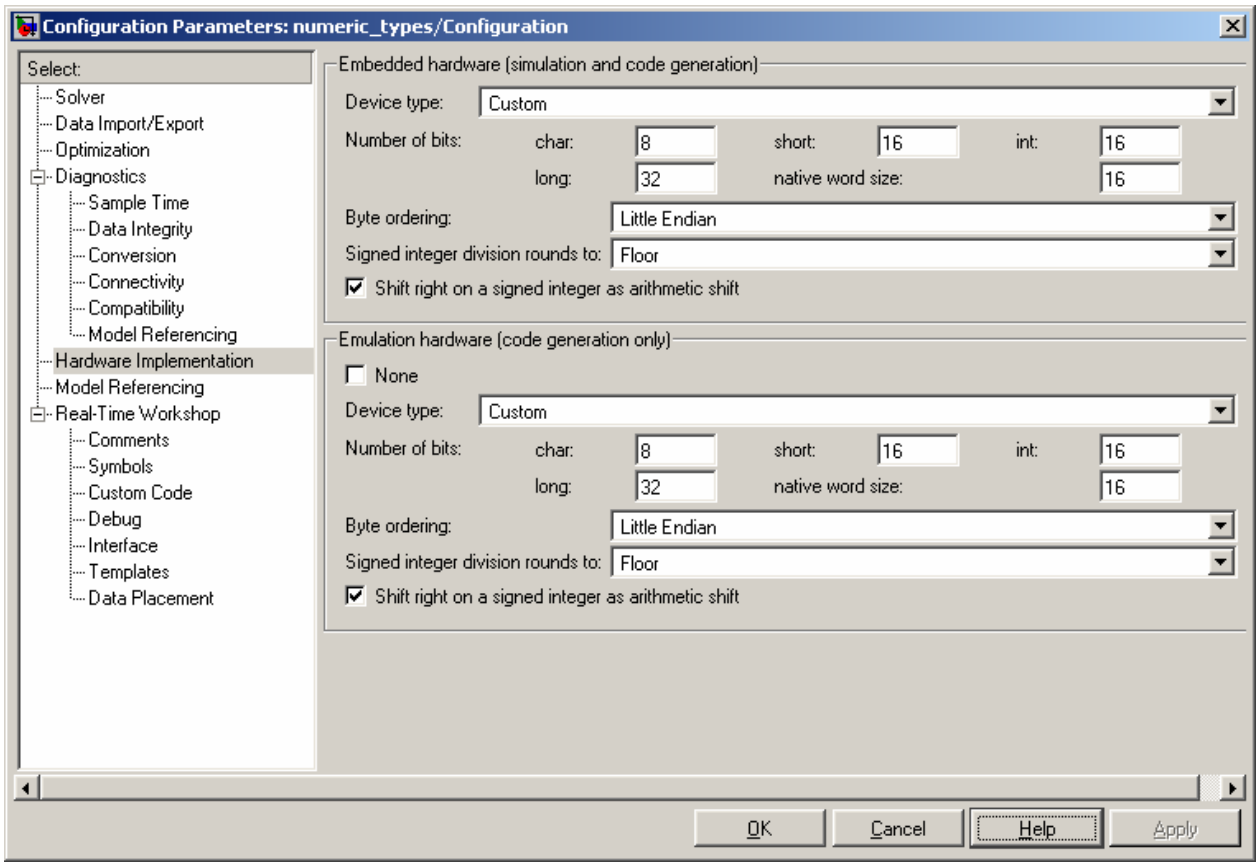


Figure 84: Hardware Implementation Pane in Configuration Parameters dialog.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 27. Managing buses (structures) with fixed-point data

Simulink provides various mechanisms for introducing structured signals and data types into the specification of a controller model and the code generated from it, such as:

- Simulink bus objects, used to strongly type bus signals and to facilitate the transfer of buses between models.
- Simulink.StructType objects, which may be used to define arrays of structures.
- Structured storage classes, which may be used to specify that signals distributed throughout a model are packaged in a structure in the generated code.

A Simulink.Bus object is created via the MATLAB command prompt, or the Model Explorer or the Simulink Bus Editor. A bus object has a set of elements, of Simulink.BusElement type. Each element defines its name, data type, complexity, dimensions, and sampling mode. The data type of an element may specify the name of another bus object, in order to create a hierarchical bus (nested structure).

The Simulink Bus Editor allows you to change the properties of bus type objects, i.e., instances of Simulink.Bus class. You can open the Bus Editor in any of the following three ways:

- Select **Bus Editor** from the model editor's **Tools** menu as shown in Figure 85.

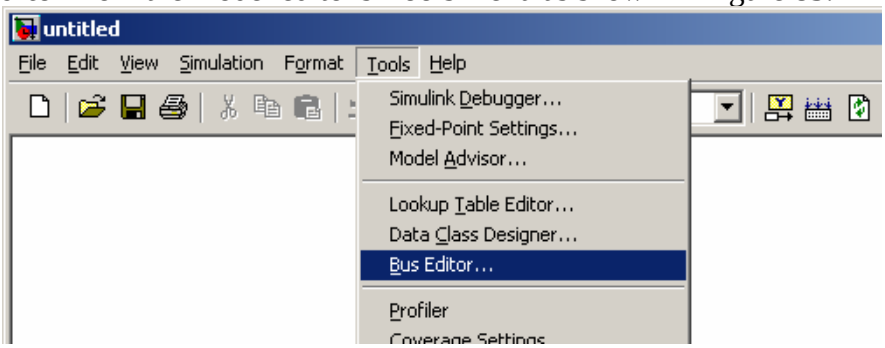


Figure 85: Launch Bus Editor from the Tools menu.

- Select the **Launch Bus Editor** button on a bus object's dialog box in the Model Explorer as shown in Figure 86.

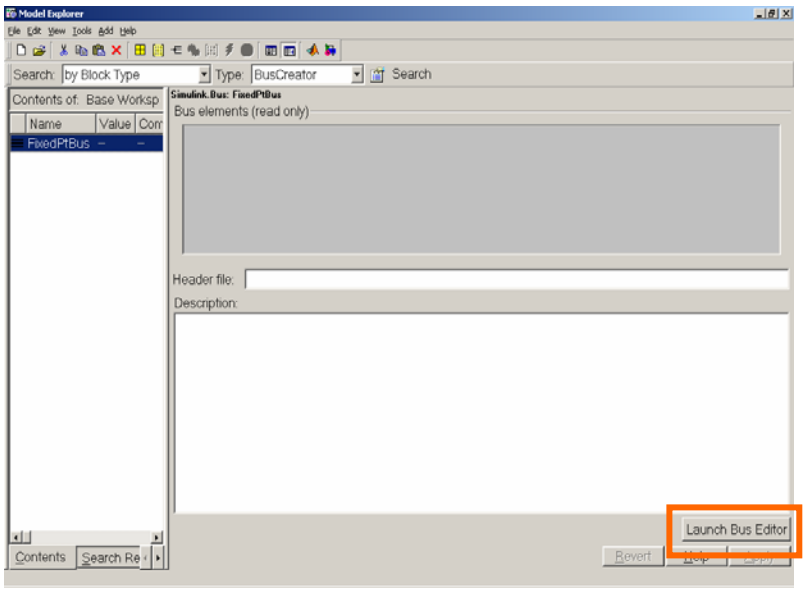


Figure 86: Launch Bus Editor from the Model Explorer after selecting the Bus Object.

© COPYRIGHT 2005 by The MathWorks, Inc.

- Enter `buseditor` at the MATLAB command line.

After you have performed one of these actions, the Bus Editor appears. A bus can be created using a bus object or a bus creator. Once a bus is created, the elements of the bus must be added. To specify a fixed-point data type for an element in a Bus, a Simulink.NumericType object can be used as shown in Figure 87. For additional details on Simulink.NumericType, refer to "Easily specify scaling and data type" in this document or to the Simulink and Real-Time Workshop Embedded Coder documentation. In this figure, CrankAngle is defined as containing a word length of 16 bits, slope (scaling) of 2^-10 and no bias.

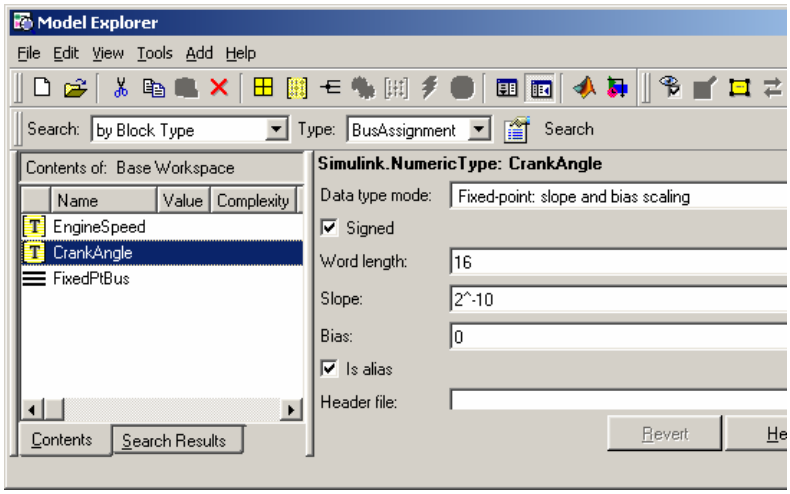


Figure 87: Create fixed-point data type named CrankAngle using Simulink.NumericType object.

You can use this newly created fixed-point data type (CrankAngle) ed to specify the data type of a fixed-point bus element (Degrees) of a bus (FixedPtBus) as shown in Figure 88. Note that only the built-in types appear as the data type choices in the pull-down list of the Data/Bus Type field of the Bus Editor. To use your own data type (CrankAngle), enter your data type manually in the Data/Bus Type field.

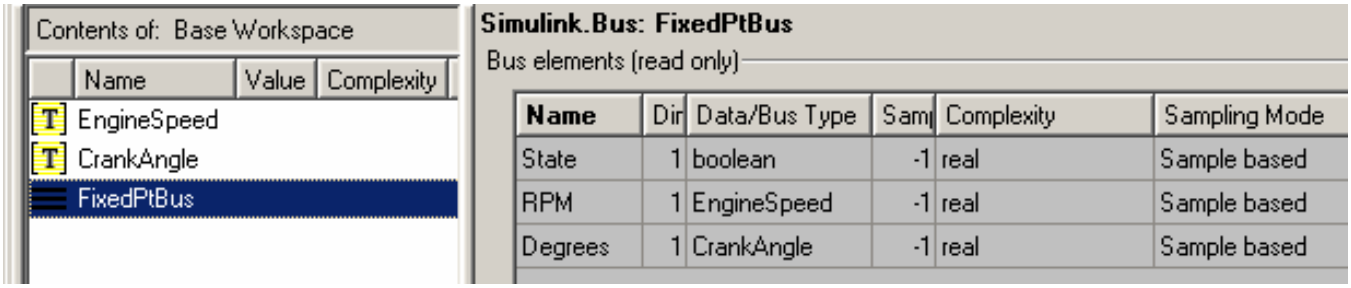


Figure 88: In Bus Types Editor use CrankAngle to specify fixed point "Data/Bus Type" for bus element Degrees in FixedPtBus.

© COPYRIGHT 2005 by The MathWorks, Inc.

To understand the above concepts, take a look at an example procedure of creating a bus in Simulink to realize a structure in the generated code. This example uses the Simulink.NumericType to create a fixed-point data type as discussed earlier:

```
typedef  struct {
  boolean      State;
  RPM            EngineSpeed;      /* typedef uint16_T RPM; */
  CrankAngle   Degrees;          /* typedef int16_T CrankAngle; */
} FixedPtBus;
```

Procedure:

1. Create data types RPM and CrankAngle using Simulink.NumericType objects. From the Model Explorer menu, **Add -> Simulink.NumericType** and edit fields as shown in Figure 89 and Figure 90.

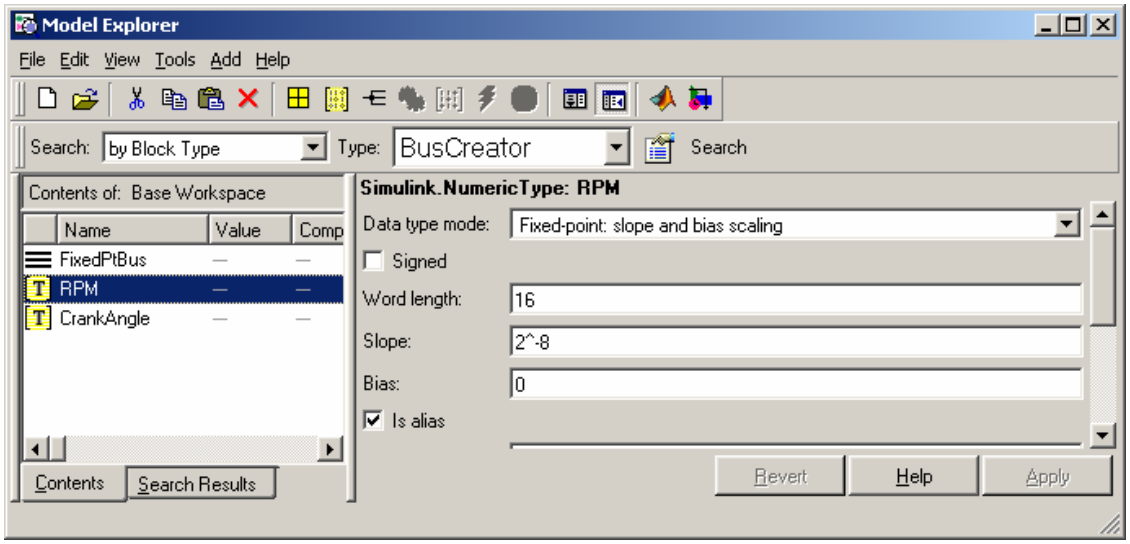


Figure 89: Create fixed-point data type RPM using Simulink.NumericType.

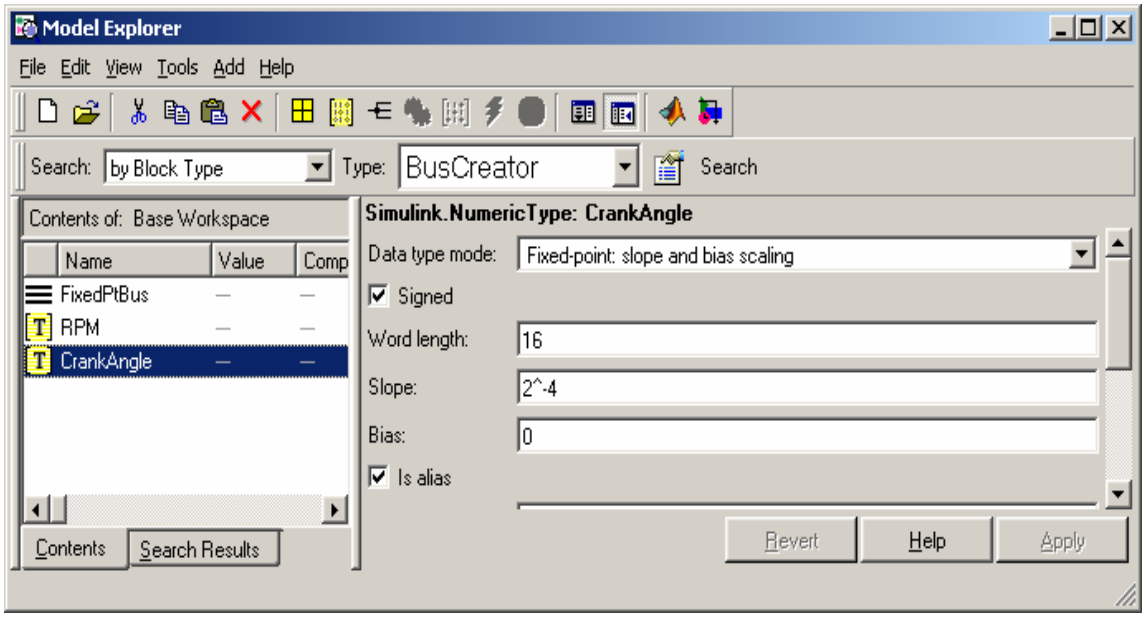


Figure 90: Create fixed-point data type CrankAngle using Simulink.NumericType.

© COPYRIGHT 2005 by The MathWorks, Inc.

2. Create a bus object named FixedPtBus. From the Model Explorer menu, Add -> Simulink.Bus and change the name from Bus to FixedPtBus.

3. Add bus elements to the FixedPtBus as shown Figure 91.

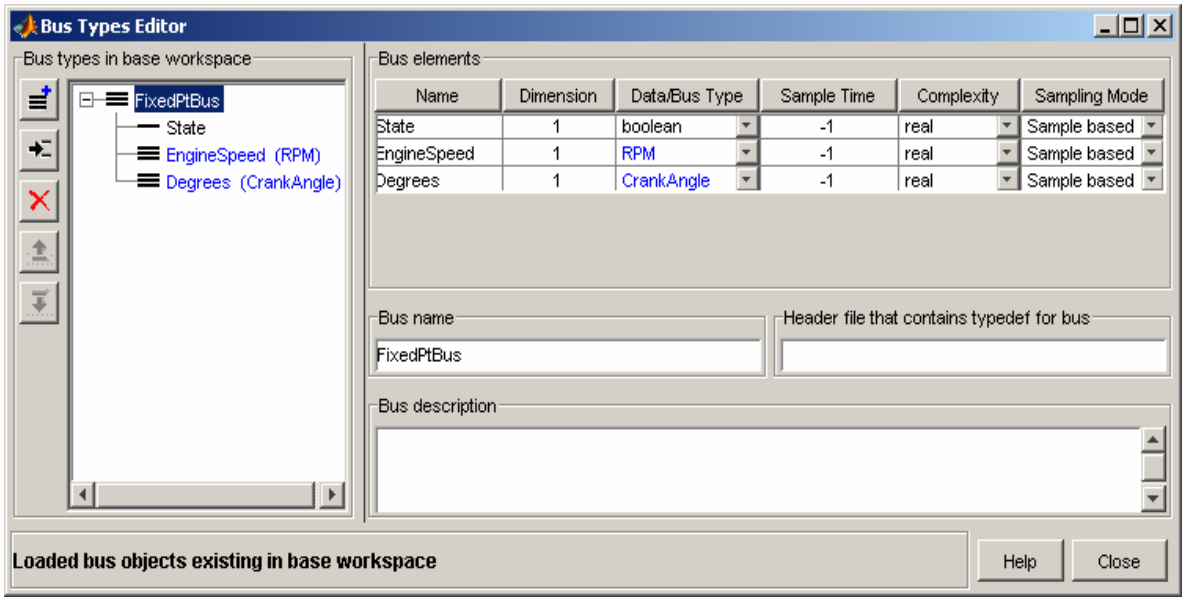


Figure 91: Edit FixedPtBus structure in Bus Types Editor.

4. Create a model as shown in Figure 92.

```
>> FixedPtBus
```

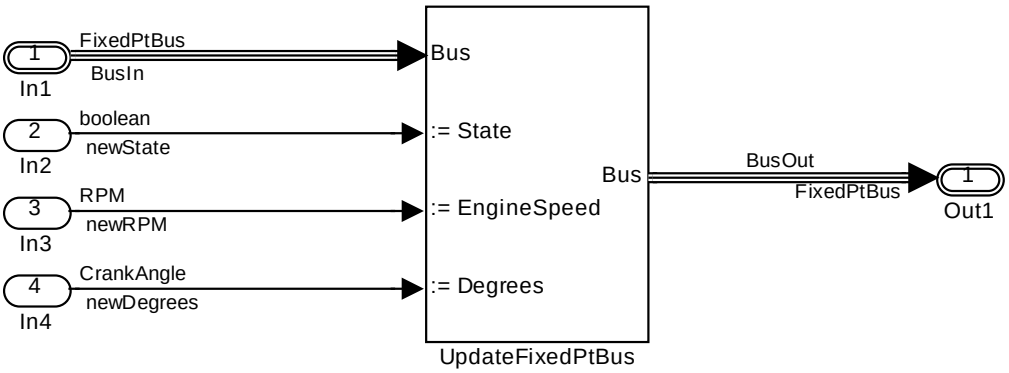


Figure 92: Model FixedPtBus.mdl.

5. Generated code and observe the code for FixedPtBus structure, RPM and CrankAngle data types in fixedptbus.c and fixedptbus_types.h files.

For additional details on Simulink.Bus, Simulink.BusElement, Simulink.StructType, structured storage classes, and bus editor, refer to Simulink and Real-Time Workshop Embedded Coder documentation**.**

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 28. Use Model Advisor to optimize model

The Model Advisor allows you to quickly analyze a model for code generation and identify aspects of your model that impede production deployment or limit code efficiency. For example, it can identify

- Questionable fixed-point operations
- blocks that generate expensive saturation and rounding code

The Model Advisor can be launched in several ways:

- From a Simulink model, choose **Model Advisor** from the **Tools** menu to start it for that model as shown in Figure 93. From Model Explorer, select the node **Advice** for model in the Contents pane to start Model Advisor for that model.

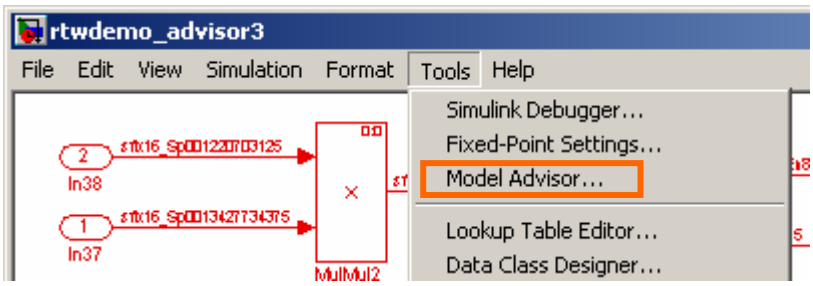


Figure 93: Launch Model Advisor from the Tools menu.

- From the MATLAB prompt, typing `modeladvisor('model')` invokes the tool for the root system of a model. If the specified model is not currently open, Model Advisor opens it. `modeladvisor(`model/system')` invokes the tool for the specified system. If the specified model is not currently open, Model Advisor opens it.

- You can also invoke the Model Advisor for models and systems using the built-in Simulink `bdroot`, `gcb`, and `gcs` handles. For example, `modeladvisor(gcb)` invokes the Model Advisor for the currently selected block (which would normally be a subsystem).

For additional details on Model Advisor, refer to Simulink documentation**.**

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 29. Reduce workload using the autoscale blockset tool

You can use the **Fixed-Point Settings** in the Simulink Tools menu as shown in Figure 94 to perform autoscaling on all data in the Simulink and Stateflow model. Autoscaling is the automatic selection of scaling values of signals and parameters based on the changing values of each signal during a simulation run. Since every signal ultimately relates back to an input port, we recommend that the input vectors have enough range to exercise all the signals in the model appropriately. This avoids having to set data type and scaling manually for each signal, and having to repeat this process until you obtain the desired results.

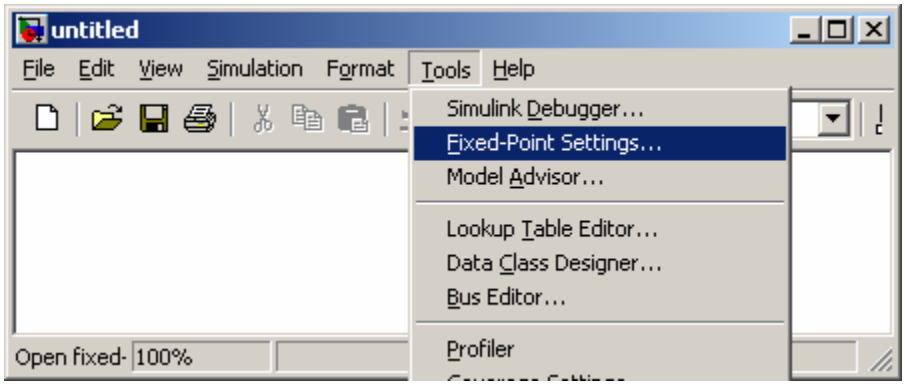


Figure 94: Launch Fixed-Point Settings from the Tools menu.

Clicking the **Autoscale Blocks** button on the Fixed-Point Settings dialog box as shown in Figure 95 automatically changes the scaling for each block that does not have its scaling locked. The tool uses the maximum and minimum data obtained from the last simulation run to log data to the workspace. If the maximum and minimum data cover the intended range of your design, the autoscaling tool changes the scaling such that the simulation range is covered and the precision is maximized.

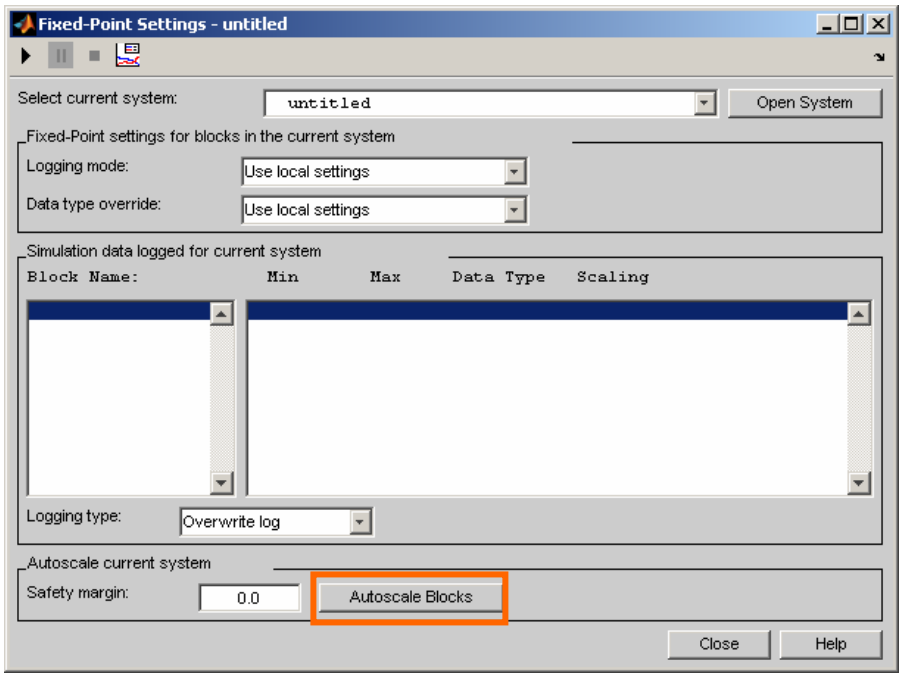


Figure 95: Fixed-Point Settings dialog

© COPYRIGHT 2005 by The MathWorks, Inc.

To obtain meaningful results from the autoscaling tool, the maximum and minimum simulation data used by the tool must exercise the full range of values over which your design is meant to run. Therefore, the simulation you run prior to using the autoscaling tool should simulate your design over its full operating range. The autoscaling tool changes scaling only for those blocks for which you select `Specify via dialog` in the **Output data type mode** field on the Block Parameters dialog.

It is especially important that you select inputs with appropriate speed and amplitude profiles for dynamic systems. The response of a linear dynamic system is frequency dependent. For example, a bandpass filter will show almost no response to very slow and very fast sinusoid inputs, whereas the signal of a sinusoid input with a frequency in the passband will be passed or even significantly amplified. The response of nonlinear dynamic systems can have complicated dependence on both the signal speed and amplitude. For such reasons, practical knowledge of the intended use of your design is the best basis for selecting inputs to exercise your system. Even with well-selected inputs, however, it is often good engineering practice to add a safety margin. If you use the RangeFactor variable as described below, the autoscaling tool can set the binary points so that an even larger simulation range is covered. A larger range reduces the chance of an overflow occurring. However, increased range results in reduced precision, so the safety margin you choose must be limited.

To reduce your workload using the autoscale blockset tool:

1. Determine the appropriate minimum and maximum values for each input signal in the model.
2. Open the model.
3. Using the Signal Builder in Simulink Sources of the Simulink Library, configure the input vectors for each signal, taking into account these minimum and maximum values.
4. Use the Fixed-Point Settings to perform autoscaling on all signals in the Simulink model.
5. Generate code.

For additional details, refer to “Automatic Scaling” section in the Simulink Fixed Point documentation. Also, see documentation on `fixptbestprec` and `fixptbestexp`.

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 30. Change preferences of blocks in a model globally

You can change the preferences of each block of a model one-by-one, by making the desired settings on the Block Preferences dialog box. This can be unnecessarily tedious. An easier way to change preferences globally or a few blocks at a time is to use the Model Explorer. The Model Explorer allows you to search the entire model by Property Value, by Property Name, by Block Type, for Library Links, by Class, for Model References, for Fixed Point, by Dialog Prompt, and by String.

You can change modifiable properties displayed in the Contents pane (e.g., a block's name) by editing the displayed value. To edit a displayed value, first select the row that contains it. Then click the value. An edit control replaces the displayed value (e.g., an edit field for text values or a pull-down list for a range of values). Use the edit control to change the value of the selected property. To assign the same property value to multiple objects (to select all use Ctrl+A, or to select a subset use Ctrl+ select objects one by one) displayed in the Contents pane, select the objects and then change one of the selected objects to have the new property value. The Model Explorer assigns the new property value to the other selected objects as well. For details on how to use the Model Explorer, refer to Simulink documentation.

Figure 96 shows a result of search (of a model using the Model Explorer), by Property Name with property name set to SaturateOnOverflow

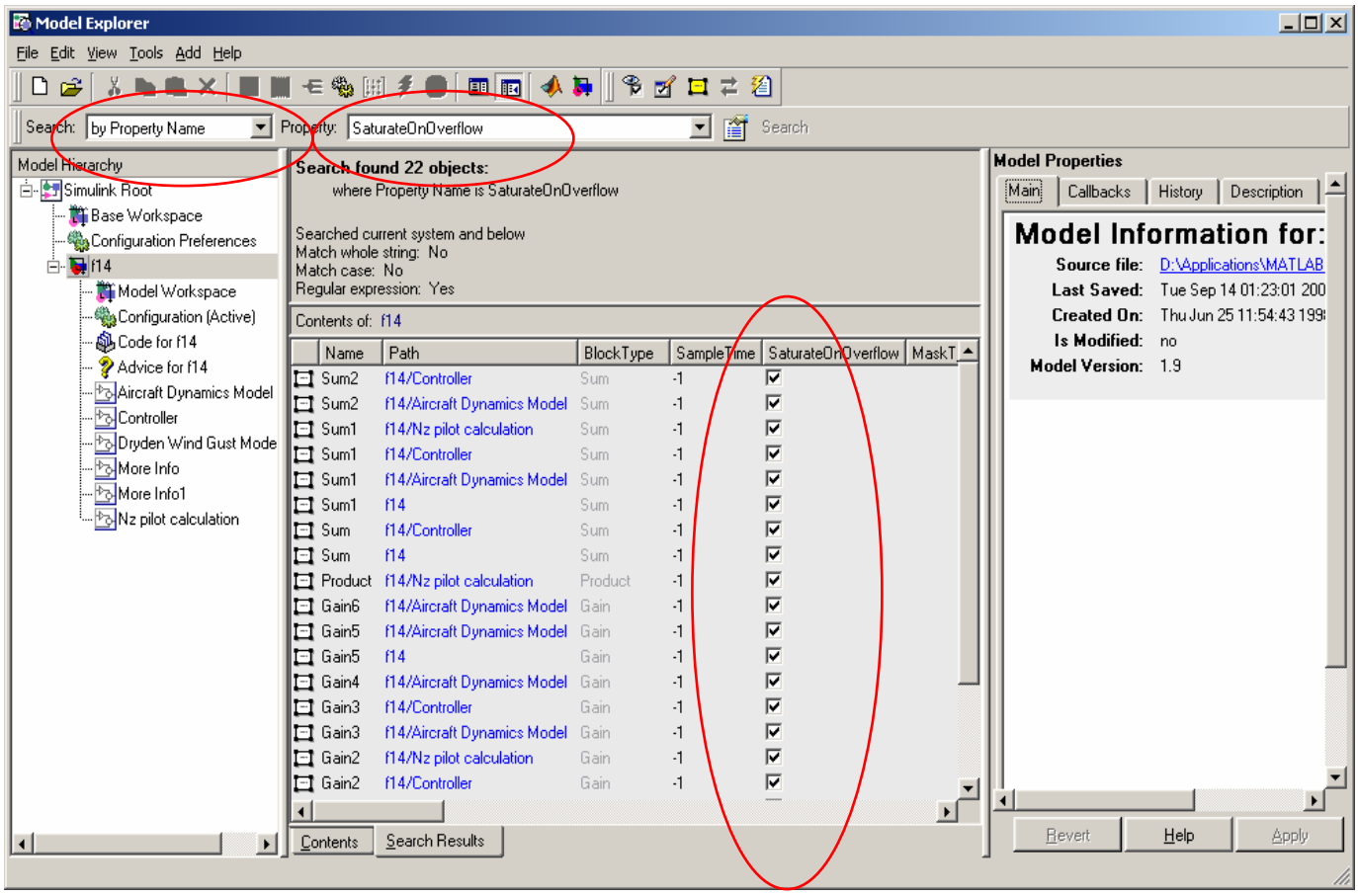


Figure 96: Model Explorer view of search results

© COPYRIGHT 2005 by The MathWorks, Inc.

Tip 31. Enforce type compatibility of parameters between workspace and model

A context-sensitive parameter is a workspace variable that is used by Simulink but it inherits its data type from the context in which it is used in the model. It is easiest to understand context-sensitive parameters in terms of their data type in the generated code, though the concept is equally relevant in simulation.

Context-sensitive rules of parameters:

1. All tunable parameters with default data type (i.e., data type is `double`) are treated as "context-sensitive." That is, the data type used for these parameters is determined by its use in the model. Data type of parameters cab be determined by typing the following command at MATLAB prompt: `class(x),` where x is the name of a parameter.
2. Parameters with values typed to other than `double` are not "context-sensitive" in the model. In other words, the data type that they are defined as in the generated code will not be derived from the use in the model; rather it will be derived from the class or type of the value. A run-time type cast will be added in the generated code as necessary.
3. All inlined parameters (not tunable) will be treated as "context-sensitive."

Case 1 Non-context-sensitive: When the parameter is set as a1 = single(0.9).

```
>> non context sensitive
```

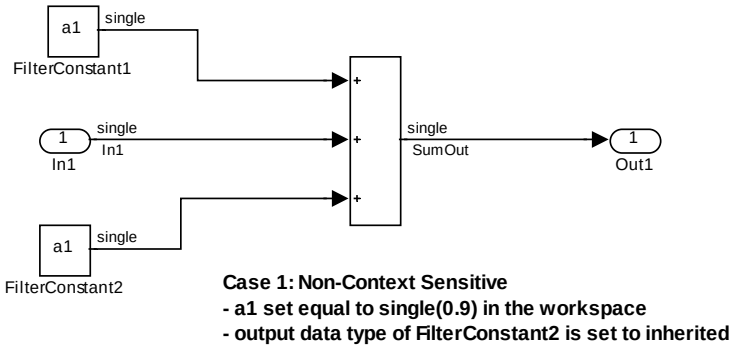


Figure 97: Model non_context_sensitive.mdl.

```
real32_T In1;

real32_T SumOut;

real32_T a1 = 0.89999998F;

void non_context_sensitive_step(void)
{
  SumOut = (a1 + In1) + a1;
}
```

Figure 98: Code for non-context-sensitive

© COPYRIGHT 2005 by The MathWorks, Inc.

Case 2 Context-sensitive: When the parameter is set as a1 = 0.9.

```
>> context sensitive
```

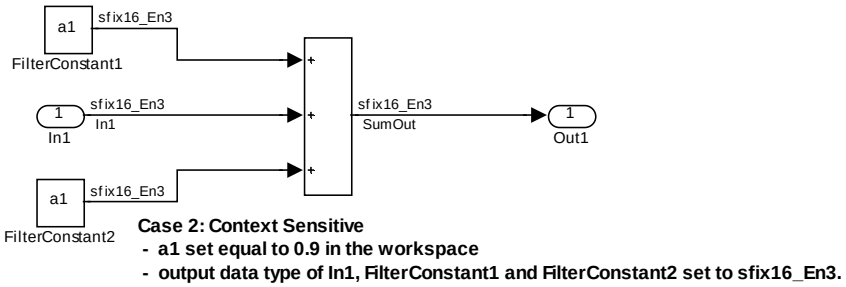


Figure 99: Model context_sensitive.mdl

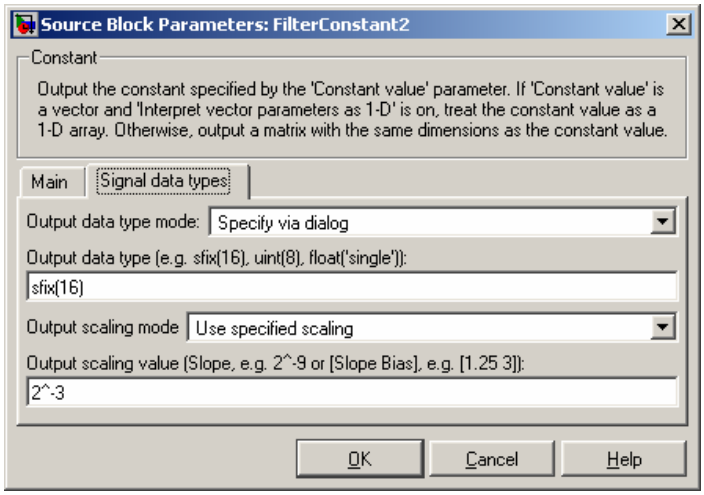

Source Block Parameters: FilterConstant2
Constant
Output the constant specified by the 'Constant value' parameter. If 'Constant value' is a vector and 'Interpret vector parameters as 1-D' is on, treat the constant value as a 1-D array. Otherwise, output a matrix with the same dimensions as the constant value.
Main
Signal data types
Output data type mode: Specify via dialog
Output data type (e.g. sfix(16), uint(8), float('single')):
sfix(16)
Output scaling mode Use specified scaling
Output scaling value (Slope, e.g. 2^-9 or [Slope Bias], e.g. [1.25 3]):
2^-3
OK
Cancel
Help

Figure 100: Signal data types dialog for FilterConstant2

```
int16_T In1;

int16_T SumOut;

int16_T a1 = 7;

void context_sensitive_step(void)
{
  SumOut = (a1 + In1) + a1;
}
```

Notice how the value of a1 is automatically converted to its equivalent fixed-point value in the code. The value of a1, set to 0.9 in the workspace, is converted to 7 (int16_T a1 = 7), its equivalent fixed-point value based on its scaling of 2^-3.

Figure 101: Code for context-sensitive

This tip illustrates the use of context-sensitive parameters. To ensure that the correct data type is used in the generated code, the following practices will be of value:

1. To make a block control a parameter's data type (that is, context-sensitive), do not cast the value of the parameter. If you do not cast the value, it is data type is `double` by default.
2. Select **Inline parameters** and then **Configure** to make the parameter tunable by adding it to *Global (tunable) parameters.* Set the *Storage class* as desired, for example, `ExportedGlobal`.
3. In the **Output data type mode** field of the **Signal data types** tab, specify the desired data type.

Note: If the same tunable parameter is used in different contexts, then it is not possible to create a single data type definition to handle different contexts. For example, **a1** cannot be used as both sfix16_En3 and single in the same model. Once the parameter is made tunable, it can only be used in one context, sfix16_En3 as shown in Figure 99.

© COPYRIGHT 2005 by The MathWorks, Inc.

Conclusion

There are many challenges faced when manually programming in fixed-point code. Similarly, care is required in automatically generating fixed-point code. The MathWorks has developed tools that enable you to find and understand proven remedies to these difficulties discussed in the “General Solution” section. However, the core of this document is the tips. These are the result of the combined experience of MathWorks developers and experienced users of MathWorks products. The systematic application of these tips, where recommended, will produce optimum code.

MATLAB, Simulink, Stateflow, Handle Graphics, Real-Time Workshop, and xPC TargetBox are registered trademarks of The MathWorks, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders.